AD-A088 601 STANFORD RESEARCH INST MENLO PARK CALIF F/G 15/3
A PROVABLY SECURE OPERATING SYSTEM.(U)
JUN 75 P G NEUMANN, L ROBINSON, K N LEVITT DAAB03-73-C-1454

UNCLASSIFIED NL

1 OF 4
AD A
088601

2

Final Report *13 June 1975*

A PROVABLY SECURE OPERATING SYSTEM

By: P. G. NEUMANN *(Principal Investigator)*
L. ROBINSON, K. N. LEVITT,
R. S. BOYER, and A. R. SAXENA

Prepared for:

USAECOM

CONTRACT DAAB03-73-C-1454

SRI Project 2581

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

9

Final Report,

11

13 June 1975

12

6

A PROVABLY SECURE OPERATING SYSTEM

10

By: P. G. NEUMANN (Principal Investigator)
L. ROBINSON, K. N. LEVITT,
R. S. BOYER, and A. R. SAXENA

Prepared for:

USAECOM

15

CONTRACT DAAB03-73-C-1454

SRI Project 2581 ✓

ABSTRACT

This report summarizes work to date toward the development of a provably secure operating system. Discussed here are:

- (1) a methodology for the design, implementation, and proof of properties of large computing systems,
- (2) the design of a secure operating system using this methodology,
- (3) the security properties to be proven about this system,
- (4) considerations for implementing such a system, and
- (5) an approach to monitoring security and performance.

Letter on file

Dist

A

CONTENTS

No. of
Pages

APPENDICES

LIST OF ILLUSTRATIONS

TABLES

INDEX OF TERMS
(Terms and Page Numbers)

PREFACE

The reader should begin this report with the technical summary that follows and then proceed to Chapter 1, which provides a more detailed overview of the report. The reader concerned with the system design will find Chapters 2, 4, 5, and 7 of primary interest, along with Appendices A and B. The reader more interested in the methodology may wish to concentrate initially on Chapter 3. The reader concerned with the statement of security properties will wish to read Chapter 8, probably first reading Chapter 3, and skimming Chapters 4, 5, and 6. Appendix C provides an illustration of an application of the system and shows that the methodology is also extendable to such applications. Although the material in the appendices is not in final form, its inclusion is essential for a thorough appreciation of the methodology and the system design.

ACKNOWLEDGMENTS

We wish to acknowledge the contributions of various people who have contributed in many ways to the work described here. Professor Robert S. Fabry was of great assistance in the first stage of design, and provided significant useful experience. John H. Wensley also contributed to the system design. Dr. Robert E. Shostak contributed to the formalization of the security properties in Chapter 8. We are very grateful to Jack Goldberg, our Group Manager, for his support as well as for his constructive criticism. We thank Dr. Theodore A. Linden for his steadfast encouragement and enthusiasm, as well as for his technical influence on the project. We also wish to express our gratitude to Mildred A. Kelley and Norinne Cox for their skills and good nature in providing secretarial and moral support.

A DESIGN FOR A PROVABLY SECURE OPERATING SYSTEM: TECHNICAL SUMMARY

Introduction

This document summarizes work to date on the development of a design for a general-purpose computing system intended for secure operation. The design facilitates the formal verification of properties about system behavior, including those defining system security.

The results of the work provide significant innovations in the following areas:

- A methodology that greatly facilitates the development of computer systems in general, and that also facilitates the formal statement and proofs of relevant system properties.
- A design for a secure computer operating system. This structured design, which has been formally specified, appears to lend itself to various efficient and reliable system implementations in hardware and software, and permits widely varying user interfaces. It also accommodates system initialization, fault recovery, and the monitoring of security and performance, relating these system aspects to the structure of the system.
- A system design that is intended to support stringent security requirements. Special security environments can be efficiently implemented that cannot be so implemented in existing systems.

Future work is needed in various areas to demonstrate the feasibility of this approach. The following areas are identified here:

- Prove the validity of specific system properties relating to system security.
- Exhibit the design of specific efficient hardware implementations to support the operating system, and explore the possibilities of retrofitting these implementations onto existing hardware.
- Show how specific secure subsystems and applications can be efficiently realized, using the operating system.

The Methodology

The methodology used in this work is addressed at unifying the various phases of system development. As it exists today, it facilitates design, implementation, verification, and monitoring. It integrates these phases, using a unified design medium and a common assertion language. The methodology provides two dimensions of structure: a structure for the system, and a structure for the system development. Both dimensions, a hierarchy of system functions and a staged development, seem to simplify the development and operation of systems.

The first dimension of structure involves a hierarchy of system functions, with a collection of related functions at each level. The functions at one level depend for their implementation exclusively on lower-level functions. Each level in the hierarchy is called an "abstract machine," and is specified independently from other levels. Hierarchical structure in a system contributes to the understandability and manageability of the development. It also facilitates system recovery, reliability, initialization, and monitoring, because each of these tasks can then be distributed and made hierarchical. Although hierarchical structures have been used previously, this work is the first attempt to formalize such structure on a large scale. In particular, it uses formal specifications to describe the desired behavior of each system function, and an abstract assertion language in which to express desired properties of overall system behavior.

The second dimension involves the partitioning of the design and implementation phases into five stages, progressing in roughly sequential order from a loosely defined system structure (i.e., functional decomposition) to a completely specified implementation. This staging encourages decisions to be made as they are needed, in a reasonable order, and at a time appropriate to the development. It permits considerable overlapping of effort, with different portions of the effort able to proceed in parallel at different stages. It also contributes to the understandability and manageability of the development. Associated with each stage are assertions of properties stated in an assertion language common to all stages. Proofs at each stage provide incremental confidence in the appropriateness of the design and implementation.

This approach simplifies the statement of system properties in abstract form. It also simplifies the proof of these system properties, since it reduces the proof to a collection of relatively much simpler proofs. In this way, it seems feasible to prove meaningful properties of a system developed according to the methodology. In the work described here, properties relating to security are of primary interest.

The methodology also facilitates incremental proofs following system alterations. This approach avoids having to repeat the entire proof effort following system alterations, requiring only new proofs of those properties that are affected by the alterations.

The methodology unifies and uses to advantage many concepts currently being considered in efforts to approach software on more of an engineering basis. Both the hierarchical structure and the staging of the development are expected to facilitate debugging, testing, operating, monitoring, optimizing, tuning, maintaining, and long-term system evolution.

The System Design

The methodology outlined above has been applied to the design of a general-purpose operating system whose security properties can be formally proven. This operating system design is represented by formal specifications for 11 levels, level 0 to level 10, level 0 being the most primitive. The functions at each of these levels are summarized in Table 1. Many of these functions are accessible only to a few immediately higher levels. Those that are accessible above level 10 constitute the user-visible operating system interface.

Table 1

OVERVIEW OF SYSTEM STRUCTURE

Level	Function
12	Commands
11	User environments
10	User process management (scheduling, interprocess communication, and synchronization)
9	Directory search management and linker (name-space manager)
8	Linkage table management (providing fast access via capabilities after initial symbolic access)
7	Symbolic object management (assuring that symbolically named objects have directory entries)
6	Directory management (mapping symbolic names to capabilities)
5	Creation of extended types and extended-type objects
4	Management of virtual storage objects (segments) and revocation
3	Physical storage management (for all on-line storage) and input-output
2	Process dispatching and system event management
1	Effective address calculation, indirection and indexing
0	Creation of capabilities, recognition of interrupts, address interpretations, and the remaining primitive machine operations

The main structures available to system users are objects of various types. These include memory objects (segments) and directories, the latter providing catalogues for symbolically named objects. Each type has associated with it a set of rules for use and protection of objects of that type. All objects of any particular type are treated uniformly, and are maintained by a collection of programs called a type manager. Creation of new types and their corresponding type managers is facilitated by the extended-type manager.

In the operating system design, all objects are referenced by means of capabilities. A capability is a protected piece of data which refers to a particular object. Each capability is protected in the sense that it cannot be modified or forged; new capabilities are obtainable only from the system (level 0). Each capability contains protection information (access rights) which indicates how the corresponding object may be used. This protection information may not be altered, although a new capability may be created (again at level 0) with fewer access rights than the original capability. Interpretation of the access rights for a given object is done by the type manager for that object. A type manager has exclusive control over the interpretation of access rights for objects of the corresponding type. In all cases, possession of an appropriate capability is necessary for access to an object.

The design as specified provides a framework for a family of operating system implementations. The basis for the operating systems is found in the mechanisms for virtual memory (level 4), capability creation (level 0), and capability recognition. The system design lends itself to efficient implementation in various ways, by supporting in hardware those functions most frequently used, independent of the level at which they appear. For example, some functions at level 10 (procedure calls) along with many functions at levels 2, 3 and 4 (including segment accessing), and all functions of levels 0 and 1, can be realized primarily in terms of single hardware instructions. Functions can also be implemented as microprograms. Thus there is a flexibility in how the system design may be implemented, with various strategies possible for constructing new hardware and for retrofitting the operating system onto existing systems.

System Security

The desired behavior of the system with regard to security (or other properties) can be formally stated and proven according to the methodology. The use of capabilities as the basis of the mechanism for accessing all objects contributes to the understandability of both the assertions and their proofs.

There are two types of assertions describing the desired security, those characterizing what the system must not do, and what the system must do. Both are necessary for satisfactory operation.

For the system under consideration, these two types of security assertions are *invariant assertions*, which specify state information that must not change, and *variant assertions*, which specify exactly how the protection state of the system must change. Proof involves showing the consistency of these assertions with the specifications.

With respect to the visible interface at any level of the system (e.g., the user-visible interface above level 10), two basic principles describe what the system may not do:

P1: There shall be no unauthorized alteration of information
(the Alteration Principle)

P2: There shall be no unauthorized acquisition of information
(the Detection Principle)

In this system, "authorization" has meaning only with respect to a mapping from capabilities to the information in the system. Then the authorization to access a particular piece of information implies having access to an appropriate capability (itself obtained in an authorized way). Authorized passage of capabilities is strictly limited to two mechanisms, having a capability placed by another user in an object for which a capability is available, or being passed a capability in a call to or a return from a procedure. (Note that the return from creating an object provides the capability for that object.) In both cases this passage is itself controlled by capabilities. As in other systems, a user's authorization thus depends largely on what is made available initially to the user at login. The identification and initial authorization are thus critical to security, but are beyond the scope of the operating system design--apart from the fact that this initial authorization can itself be made secure as a part of the operating system. (Numerous new developments seem to make the identification problem tractible.)

For each relevant system function, P1 and P2 are stated formally in terms of two corresponding invariant assertions involving capabilities. In addition to these invariant assertions for security, there are formally stated variant assertions. For each function in the system these variant security assertions define the precise meaning of acquisition and alteration as effects upon the security state, in terms of access to capabilities (and therefore to their corresponding objects).

There are two additional security principles, which are successively harder to state formally. The first of these is the Guaranteed Service Principle:

P3: There shall be no unauthorized denial of service.

This principle states that users should never be denied access to objects (i.e., use of resources) to which they are entitled. For example, an error in a scheduling algorithm might prevent certain programs or processes from ever executing. Many meaningful cases of guaranteed service can be formally covered by specific assertions (e.g., regarding fairness of resource allocation) and proven as for P1.

The second additional security principle refers to preventing unauthorized leakage of information by inference on visible interfaces. Such leakage may occur from the system to a user, or from one user to another. In certain cases it is possible to deduce (e.g., by repeated statistical sampling) certain information (or properties of it) whose acquisition is otherwise impossible. This can be stated intuitively by the following Confinement Principle:

P4: There shall be no unauthorized leakage of information.

Note that the confinement principle P4 deals with the unauthorized acquisition of information, similar to P2. The difference is that P2 refers to access by normal paths (i.e., values of calls to system functions), while P4 refers to access by deductive or other inference. Although some cases of this principle can be covered by formal assertions, rigorous enforcement is unattainable in a theoretically complete sense. Nevertheless, it appears that the methodology can permit a characterization of many threats of such leakage.

The above dis ussion on system security has concerned the operating system (i.e., levels up to level 10). Given the need to develop a particular user environment on top of the operating system (e.g., levels 11 and 12), the methodology also applies to the design and implementation of such an environment, and to the statement and proof of the relevant security properties of that environment, above and beyond those already covered by the operating system. An example of such an environment is one which enforces the military classification system, with levels of classification and categories of relevance. Additional security assertions can be stated and proven that precisely reflect the requirements of such an environment. Other such special security environments that can be supported by the operating system involve systems with special interpretive authorization of access, or where different users or subsystems are suspicious of each other.

Conclusions

At present, a set of specifications exists for all functions visible at the interface to each level from level 0 to level 10, as well as for certain functions internal to these levels. Various efficient implementations seem possible. The task of proving that the system satisfies the two basic security principles P1 and P2 has been defined sufficiently to conclude that it seems feasible. Further, various complex security environments seem to be efficiently implementable. The proof methodology permits the proving of additional properties about the security of such environments.

Future work is needed in several areas to demonstrate conclusively the usefulness of the methodology and the suitability of the design. This work should include proving that the security assertions are satisfied by the specifications for the operating system, examining various strategies for hardware realization of the system, and investigating the implementation of several secure application environments.

Other work at SRI is also using the methodology to develop a hardware-fault-tolerant computing system for commercial aircraft (sponsored by NASA). Initial results are extremely promising. Additional work at SRI is aimed at extending the methodology and developing a semiautomatic verifier. Work at MITRE is using a similar approach, aimed at implementing a version of Multics that supports the military classification scheme.

Chapter 1

INTRODUCTION

The effort described here involves the design and implementation of a family of general-purpose computer systems, and formal proofs of significant security properties concerning these systems. This report summarizes the work to date. It includes a new methodology for the design, implementation and proof of large systems. It also includes the structure for and representation of a general design for an operating system and supporting hardware, thereby illustrating the use of the methodology. Specific properties about the security of the system are formally stated, and it is shown how these properties can be formally verified. It is emphasized that even if no proofs are attempted, the methodology can contribute significantly to the suitability of systems developed according to it.

Future work is needed to verify that the design satisfies the security-related assertions, to show how the design can be used efficiently for various subsystems and user environments, to show how the system may be implemented efficiently in software and hardware, and to verify that such an implementation is itself correct.

Efforts to develop large operating systems have typically been very protracted and hard to control. In addition, it has been virtually impossible to prove the correctness of meaningful system properties. The methodology described here appears to make feasible the design of realistic general-purpose secure computer systems as well as formal statements and proofs of their properties. The design framework defined here covers a family of systems. For descriptive simplicity it is thought of as including a single operating system, with various possible command interfaces and various hardware support. In fact, a collection of visible operating systems functions is specified here, although neither an actual command interface nor a specific machine implementation is specified. However, the command structure of most conventional operating systems (e.g., OS/370, Multics, MCP, SCOPE, etc.) could be implemented approximately compatible with our system design. The system design provides

- a flexible protection mechanism, supporting controlled sharing of system- and user-defined objects, and the establishment of specialized protection domains,
- an on-line storage system for all user-defined objects known to the system,
- a framework for attaining high availability and maintaining security in the presence of hardware faults, including facilities for backup and recovery for user and system objects,
- support for multi-programming and multi-processing, and
- a framework for monitoring performance and security.

Although the operating system is not described here in terms of particular hardware, essentially all security-relevant basic instructions have been specified. However, various alternatives are possible for supporting hardware, which is not limited to any particular architecture or machine. The system appears capable of competitively efficient implementation, in terms of microcode or other interpretation or existing hardware, or in terms of specially designed or modified hardware, or both.

In pursuing general goals for system operation and for proof of security, the initial research has focused on the following three problem areas:

(A1) the development of a suitable formal methodology for the design and implementation of complex computer systems, and for the statement and proof of properties of the design and the implementation;

(A2) the design and formal specification of a secure system using the methodology;

(A3) the development of a formally defined language for stating security-related assertions, and the statement of these assertions.

Work on the above areas is basically complete. From the results reported here, it appears possible to design an operating system that can be shown to satisfy stated security assertions. Further work should address

(A4) verification that the design of (A2) satisfies the security assertions;

(A5) implementation of the design of (A2), i.e., the implementation of the system on existing hardware, and/or on some hypothetical (albeit formally specified) hardware system;

(A6) implementation of typical applications, and proof of their correctness.

Section 1.2 briefly summarizes a new methodology for design, implementation, and proof. Section 1.3 summarizes the operating system design according to the methodology. Section 1.4 summarizes the approach to stating and proving security assertions. These topics are discussed in greater detail in subsequent chapters. A short self-contained description of much of the work is found in Robinson et al. [75].

1.2 The Methodology for Design, Implementation and Proof

Although many operating systems have been developed, there have been no serious attempts to state (much less to prove) explicit properties of any of these systems. In fact, there is a general skepticism about the feasibility of ever proving a large-scale operating system--because of the sheer complexity of the programs, the problems of concurrency, and the difficulties of stating assertions on what the system is intended to do. The skepticism is certainly justified with regard to contemporary systems such as OS/370 or the present version of Multics. Every existing system has had its security compromised at various times--in some cases with far-reaching consequences. Nevertheless, it appears realistic to us to prove properties of an operating system that is designed according to a methodology that facilitates such proofs. This is particularly significant when the security of the system is involved, for which proofs can be of great value. With respect to proof, the requirements for our methodology include the following:

- The proof of a system should reduce to the proofs of small programs (e.g., 30-line programs), although there may be many such programs.
- The assertions for each of these programs should be concise and relevant to the context of each program.

In addition to enhancing provability, other goals are as follows:

- The methodology should enhance the cost-effectiveness and desired behavior of the resulting systems (e.g., with respect to efficiency, reliability, security, and recovery).
- The methodology should enhance the development of each system or family of systems.

To achieve these primary goals, the methodology should:

- enhance each phase of the overall effort, including design, implementation, debugging, testing, verification, operation, fault recovery, monitoring, tuning, maintenance, and evolution of the resulting systems.

- integrate these phases--e.g., using common languages and common formalizations.

- provide successively increasing confidence throughout the development as to the appropriateness of the design and the implementation, and to the thoroughness of the verification.

- inspire good management--e.g., by simplifying the job of management, and improving the identification of parts of the work.

- support relevant human needs (of users, system programmers, maintainers, and managers).

These goals are all addressed by the methodology. This methodology uses a staged development, and is based upon decomposing the system into a hierarchy of formally specified abstract machines M_0, M_1, ..., M_n. Each abstract machine is self-contained in its specifications. M_0 is the most primitive machine under consideration. For each abstract machine M_i, a set of abstract programs $\{P_i\}$ can be thought of as interpretively executed to implement M_{i+1}. (In fact, the actual implementation may be much more efficient.) The specification method for the abstract machines is based upon Parnas' method for "module specification" (Parnas [72a, 72b, 72c, and 72d], and Parnas and Siewiorek [72]). An abstract machine M_i is characterized by its state and by operations that change its state.

The state for the abstract machine M_i is represented by a set of V-functions (Value-returning) $\{V_i\}$, which are somewhat analogous to state variables of a sequential machine, except that V-functions can have arguments over an arbitrary domain. A caller of an abstract machine can use the V-functions to learn about the state of a machine, but not to change the state. The state changes of M_i are effected by calling O-functions (Operations) $\{O_i\}$; however, a call on an O-function does not return a value. There are significant advantages to separating the function into

O- and V-functions, with respect to design and proof (see below). However, in certain cases, there is a need for an OV-function, which accomplishes a state change and also returns a value in one indivisible operation. (By being careful about the use of OV-functions, the advantages of the separation between O-functions and V-functions can be retained.)

This hierarchy of abstract machines forms the basis for the staged development. It also helps to decompose the proofs of operating system properties into proofs of small programs $\{P_i\}$, each P_i running on M_i and implementing the O-, V-, and OV-functions of M_{i+1}. The staged development involves five stages of design and implementation (S1 to S5) and five associated stages of verification (V1 to V5), as follows:

(S1) decomposition of the system into a hierarchy of abstract machines, selection of functions for each machine, and determination of which functions are available at which levels in the hierarchy.

(S2) formal specification of each function in terms of the value returned and/or the state changes. These specifications take the form of assertions in a nonprocedural assertion language.

(S3) correspondence between the state of each machine M_i and the state of M_{i-1} for $i > 0$. In particular, this stage involves writing assertions (written in the assertion language of S2), called mapping functions, that constrain the V-functions of one level relative to those of lower levels.

(S4) implementation of each of the functions of each machine M_i as a program using the functions of M_{i-1} for $i > 0$. The implementations are called abstract programs.

(S5) implementation of all primitive functions as programs in the instruction set of the hardware in some compiler or other language.

The results of stage S3 provide a design; the structure of the system, the function of each abstract machine operation, and the interrelationships among the states of the abstract machines are specified but not implemented. The results of stage S4 provide an abstract implementation, since each operation is implemented in an abstract programming language out of lower-level operations not necessarily directly supported by any hardware. The results of stage S5 are called a complete implementation, for indeed the system is capable of execution.

This staged design-and-implementation methodology is functionally oriented rather than either software driven or hardware driven. However, awareness of hardware concepts arises in the early stages (particularly at the lowest levels).

The expression "top-down" is normally used to describe successive refinement, from a high-level representation to an implementation. The methodology can be essentially "top-down" with respect to the stages of increasing implementation specificity, S1 to S5. However, backtracking to earlier stages is normal. Neither the design nor the implementation at any stage need be top-down with respect to the levels.

The verification phase of the methodology is closely integrated with the design-and-implementation phase, with five stages associated with S1 to S5.

(V1) Establishment of global assertions about the desired system behavior to be proven. One use of global assertions is to define the necessary constraints for the security of the system, as done here. (Global assertions may also be used to define requirements for reliability and recovery.)

(V2) Verification of S2: Verification that the specifications are self-consistent at each level, and that the global assertions of V1 follow logically from the specifications of S2. Note that any relations among levels (e.g., the mapping functions of S3 or the abstract implementations of S4) that are consistent with the specifications of S2 are therefore also consistent with the global assertions.

(V3) Verification of S3: Verification that the mapping functions defined in S3 are consistent with each other and with the specifications in S2.

(V4) Verification of S4: Verification that the abstract programs in S4 are correct with respect to the specifications and mappings of S2 and S3, respectively. S2 and S3 are used to derive the correctness criteria for the abstract implementation of each function.

(V5) Verification of S5: Verification that the abstract programs in stage S4 are correctly implemented in the hardware instructions (stage S5). This stage guarantees that the system assertions are satisfied by the system as implemented. Verification of the correctness of the hardware is also feasible.

The stages of verification are discussed in detail in Chapter 3, along with the associated stages for design and implementation. The statements of specific assertions about the behavior of the system, especially with regard to security, are presented in Chapter 8.

The relationships among the stages are shown diagrammatically in Figure 1. In this figure, each circle denotes an abstract machine whose interface (i.e., a set of operations visible at that level) is symbolized by an arrow. The state of each abstract machine is symbolized by a square.

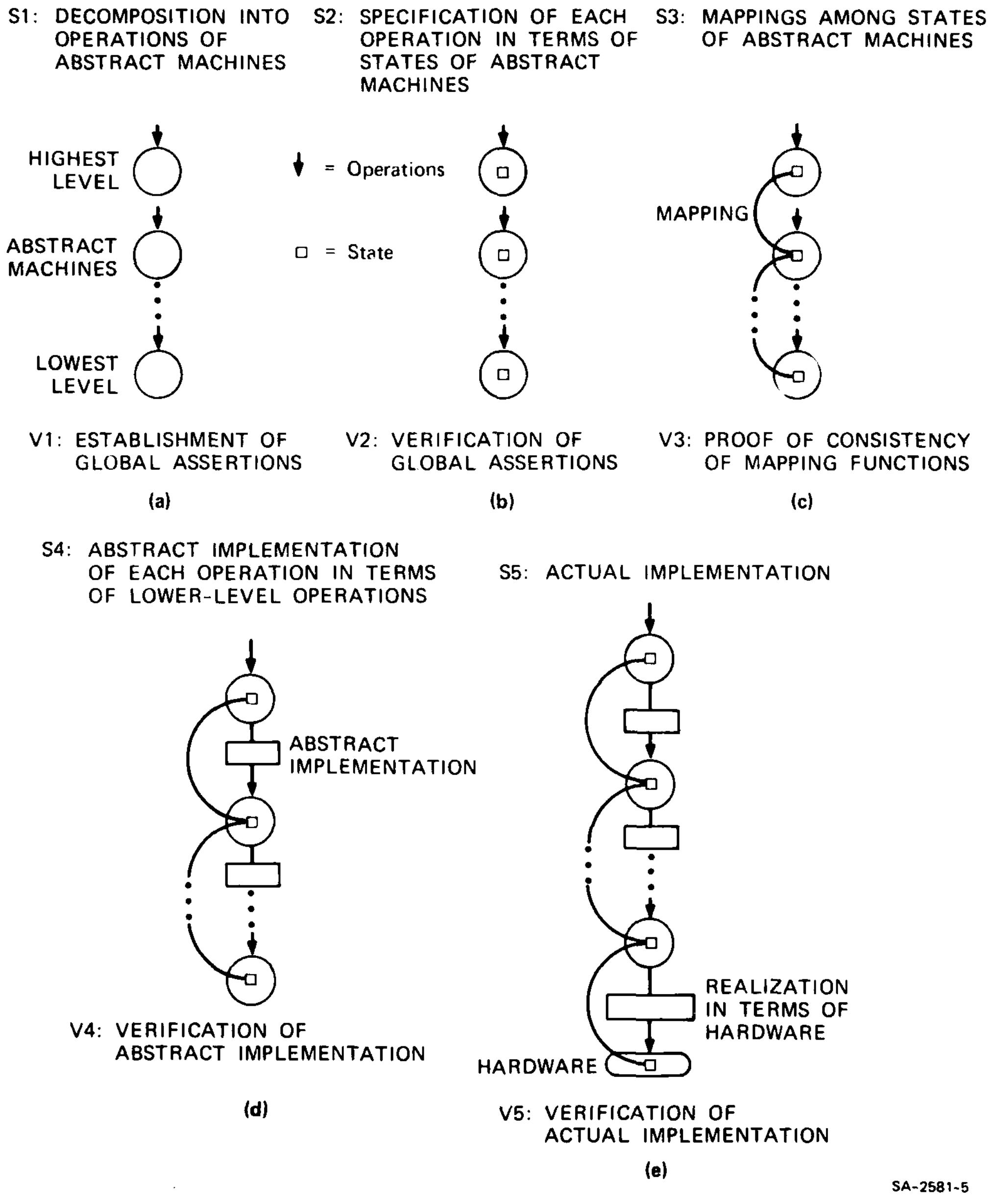

S1: DECOMPOSITION INTO OPERATIONS OF ABSTRACT MACHINES
HIGHEST LEVEL
ABSTRACT MACHINES
LOWEST LEVEL
V1: ESTABLISHMENT OF GLOBAL ASSERTIONS
(a)
S2: SPECIFICATION OF EACH OPERATION IN TERMS OF STATES OF ABSTRACT MACHINES
↓ = Operations
□ = State
V2: VERIFICATION OF GLOBAL ASSERTIONS
(b)
S3: MAPPINGS AMONG STATES OF ABSTRACT MACHINES
MAPPING
V3: PROOF OF CONSISTENCY OF MAPPING FUNCTIONS
(c)
S4: ABSTRACT IMPLEMENTATION OF EACH OPERATION IN TERMS OF LOWER-LEVEL OPERATIONS
ABSTRACT IMPLEMENTATION
V4: VERIFICATION OF ABSTRACT IMPLEMENTATION
(d)
S5: ACTUAL IMPLEMENTATION
REALIZATION IN TERMS OF HARDWARE
HARDWARE
V5: VERIFICATION OF ACTUAL IMPLEMENTATION
(e)
SA-2581-5

FIGURE 1.1 SUMMARY OF THE METHODOLOGY FOR COMPUTER SYSTEM DEVELOPMENT

Note that up to stage 2, the abstract machine at each level appears to be independent of those at other levels, although the intent of eventual implementation is at least partially implicit in the decomposition of stage 1. The mappings between states are symbolized by the arcs interconnecting the states, and provide the basis for the state representation at each level in terms of lower-level states. The abstract implementations and the real implementation at stages 4 and 5 are symbolized by rectangles, relating the implementation of the operations at each level to the next lower level. (As noted in Chapter 3, this figure is somewhat oversimplified. For example, the hardware interface may be distributed over several levels, although it is symbolized in the figure at the lowest level.)

In summary, then, the methodology has the following properties:

- The methodology facilitates the choice of an appropriate hierarchical design structure that in turn contributes to simplifying implementation and proof. This also can contribute to the anticipation and simplification of optimization, maintenance, and long-term evolution of the system.
- Staging of the design and implementation phases permits unnecessary binding of implementation details to be deferred, and provides additional structure for the development of the system.
- Formal specification of the design facilities understandability and proof.
- The methodology associates proofs with the design and implementation, so that proofs of correctness are meaningful at each stage, with successive stages providing incremental confidence in the development.

- The methodology permits incremental proofs after system alterations, taking advantage of earlier proofs of previous stages and of lower levels that are unaffected by the alterations, rather than requiring proof from scratch.

There has been much work recently on design structure (e.g., Simon [62], Dijkstra [68a,b,c,72], Hoare [71,74], Parnas [74]), although there has been considerable skepticism regarding the design of hierarchical systems. The next section illustrates our design, which is hierarchical (according to the methodology). The design seems to have suffered relatively little by being constrained to be hierarchical. Thus much of the skepticism may be misplaced.

1.3 The Design of the System

The design of the operating system has followed the methodology of Section 1.2. The decomposition (stage 1) of the system into a hierarchy of abstract machines is summarized in this section. The selection of functions (completing stage 1) is given in Appendix A, along with specifications of each function (stage 2). Mapping functions (stage 3) are given in Appendix B. Specifications of the security-relevant portions of the hardware are included in Appendix A as a part of the specifications of the lowest-level functions. Abstract implementations (stage 4) are illustrated in Appendix A, Section 7. Implementation details (stages 4 and 5) are found in Chapter 7.

The Use of Capabilities

The unit of protection, sharing and naming is an object. Types of objects include segments, processes, directories, and input-output devices. New types (i.e., extended types) may be defined for which objects may be created, maintained, and deleted. The design of the operating system is based on the use of capabilities (Dennis and Van Horn [66], Fabry [67, 74], Lampson [69a, 69b], Needham [72], Sevcik et al. [72], Jones [73], Sturgis [73], and Wulf et al. [74]) to provide protection for all objects.

A *capability* is a protected piece of data that serves as both a name and address for an object. A capability is protected in that it is created by the system and cannot be modified or forged. It includes a *unique identifier* (uid) that is a nonreusable system-generated number derived from such a source as the instantaneous reading of a fine-quantum clock or a systemwide nonrecycling counter. The capability for an object also includes an *access code* (ac). The access code specifies which types of access to an object are permitted for a user presenting a capability for that object. When a capability is presented, it is interpreted by an accessing mechanism appropriate to the object referred to by the capability, namely, the *type manager* for the type of the object. Protection results from the enforcement of the rule that access to an object is permitted only upon presentation of an appropriate capability corresponding to that object.

The use of capabilities greatly aids the effort to achieve a provably secure operating system. The most notable advantages are that

- Capabilities provide a uniform and noncircumventable means of accessing and protecting all objects. Their use has a significant impact, unifying the design, and simplifying the proof process.

- Capabilities are well suited to hierarchical design, and lead naturally to the use of extended-type objects (see below) to provide layers of abstraction and protection. While the protection afforded *to* capabilities and the creation of capabilities are centralized, the interpretation of capabilities (i.e., the protection afforded *by* capabilities) is distributed as appropriate. Thus capabilities facilitate the separation of mechanism and policy.

The design differs from previous capability-based systems in that great care has been taken to structure the system hierarchically into conceptually simple modules, to use a formal design medium, and to anticipate the need for (hierarchical) proofs of correctness. System initialization and fault recovery are carefully integrated with the hierarchical structure, and proceed hierarchically. In addition, solutions are included for two traditionally difficult problems of capability-based systems:

- The revocation problem (how to invalidate a subset of the existing capabilities for an object, regardless of their whereabouts) is solved by the approach of Redell (Redell and Fabry [74]). When the need for revocation of access to a particular object is anticipated, a revocable capability is created for the object. When revocation is desired, execution of the revoking operation makes the revocable capability invalid (as well as any copies of it).
- The lost-object problem (or preventing the presence of any objects for which capabilities no longer exist) is avoided by ensuring that essentially all user-created objects have at least one valid entry in the directory system.

An Overview of the Design

A simplified outline of the design structure is now given, which includes thirteen hierarchical levels of design. The operations at each level are used only by higher levels. Beginning with the highest level (i.e., the level least dependent on the hardware), the levels are as follows:

12. Commands

11. User environments

10. User process management (scheduling, interprocess communication, and synchronization)

9. Directory search management and linker (name-space manager)

8. Linkage table management (providing fast access via capabilities after initial symbolic access)

7. Symbolic object management (assuring that symbolically named objects have directory entries)

6. Directory management (mapping symbolic names to capabilities)

5. Creation of extended types and extended-type objects

4. Management of virtual storage (segments) and revocation

3. Physical storage management (for all on-line storage) and input-output

2. Process dispatching and system event management

1. Effective address calculation, indirection, and indexing

0. Creation of capabilities, recognition of interrupts, address interpretations, and the remaining primitive machine operations.

For each of these levels, Table 1.1 provides a summary of what operations are performed, what objects are primitive, and what concepts are made invisible by that level.

Discussion begins with some of the objects provided by the operating system. The basic unit of data protection in the system is a *segment*, which is a contiguously addressable unit of virtual storage. Data and capabilities are self-identifying (e.g., tagged), and may be freely intermixed with a segment. Segment objects are maintained at level 4. The physical location of each segment is invisible above level 4. (At any one time, its physical storage residence may be on various devices, or totally on one device, at the discretion of the system.) A segment is

Table 1.1

SUMMARY OF THE OPERATING SYSTEM STRUCTURE

Level	Primitive Objects	Operations at this Level	Concepts Invisible Above this Level
12	Commands	Command interpretation	User environment formats
11	User environments	System commands and user programs	Command implementations
10	User processes, signals	Scheduling, interprocess communication	Fixed number of scheduled processes
9	User name space	Handling of linkage faults and searching	Linkage faults, linkage sections*
8	Linkage tables	Linkage table management	Linkage section implementation
7	Named objects	Use of directories and objects	Uncatalogued objects
6	Directories	Maintenance of directories and entries	Directory implementation
5	Abstract types	Creating and deleting new types, objects	Backup of objects
4	Segments, revocation	Creating and deleting segments, revoking	Memory faults, storage addresses
3	Fixed virtual memory to support level 4	Memory fault handling, storage management	Memory and I-O addresses
2	Scheduled processes	Dispatching, interrupt handling, synchrony	Processors, interrupts
1	Main-memory pages	Load, store, call, return, stack operations	Effective address computation
0	Capabilities, interrupts	Basic hardware, interrupt decoding	Hardware implementation

*Capabilities may be hidden from users by level 9, if desired.

ultimately accessed only by the presentation of a capability for it, just as is any other object, and thus a segment may be named, accessed, and protected, independently of any other segment. Creation of revocable capabilities and of revoking capabilities, and the handling of revocation itself, are also handled at level 4. Physical storage management, including page placement and replacement, is done by level 3.

Level 5 records the definition of each extended type, and handles the creation and deletion of objects of these types. The interpretation of an extended type includes its implementation in terms of segments and/or other objects. As an example, we choose to implement a directory as an extended-type object, each entry of which is an association of a symbolic name with a capability. Directories are maintained by level 6. Above this level, objects may be referred to by a symbolic name for the directory entry, rather than by a capability.

To solve the lost-object problem, essentially every user-created object is created with a symbolic name and included in some directory. To create an object, users must employ level 7, which in turn uses segment creation and/or extended-type object creation (levels 4 and 5, respectively) and directories (level 6). For each object, there is one directory entry (called a distinguished entry) that cannot be deleted unless the object has been deleted. (Distinguished entries may be moved and renamed--except those for directories.) Other entries that are not distinguished may be freely deleted.

The directory system is tree structured with respect to distinguished entries, since each directory must itself have a distinguished entry in some other directory. (The nondistinguished entries serve a purpose similar to the file system "links" in Multics.) However, several disjoint tree structures may exist simultaneously.

For system efficiency, symbolic names are used only on initial access. This is achieved with dynamic linking (cf. Multics; Organick [72]), via an indirection through a linkage table. After the first (symbolic) access, the appropriate linkage table contains a capability for the desired object, Linkage tables are maintained by level 8.

To permit the knowledge of specific directories to be invisible, a search strategy may be used (as in Multics). Searching the directory system for a given symbolic name is done by level 9 in response to an attempt to access a previously unlinked object. In this way, capabilities may be made totally invisible to users of the system.

A process may be conveniently thought of as an instance of the execution of a program by a processor (physical or virtual). Many processes may exist on the system at one time. Below level 2, there are at most as many processes executing as there are physical processors. At level 2, there is a (larger) fixed number of processes (called scheduled processes), all of which have been authorized to run by level 10 (the scheduler). At level 10, there is a variable number of processes (called user processes), createable as needed at that level. Level 10 is also responsible for communication among user processes, and for control of process synchronization.

Levels 0 and 1 provide the basic addressing mechanisms, as well as all of the logical, arithmetic, and control operations required for processing. For a highly efficient implementation, the operations of levels 0 and 1 are expected to be realized in hardware. Some higher-level operations are also expected to be implemented in or assisted by hardware.

A more complete view of the operating system design and its motivation is provided by Chapters 4 through 7 of this report. Appendixes A and B provide formal specifications and state representations for the design.

The use of capabilities as the mechanism for accessing and protecting objects in a hierarchically structured system is of great help in stating assertions about security. Since a capability is the only means of accessing an object, and since capabilities are nonforgeable (e.g., tagged by the hardware), the only way to access an object that already exists is to have a capability for the object. The next section gives an intuitive overview to the nature of security and the role of assertions about security, discussed further in Chapter 8.

1.4 The Meaning of Security

A computing system is expected to have well-defined behavior, including availability of desired facilities, correctness of operation, suitable performance, and recovery from hardware malfunctions. It should include appropriate mechanisms for isolating users altogether, except where controlled sharing of resources is desired. These mechanisms pertain, for example, to preventing the unauthorized accessing and modification of information. The question "Is the system secure?" is meaningful only with respect to precisely stated (formal) assertions defining what is meant by security. Thus there are two key steps to being able to prove anything about system security, the ability to specify formally the design of a system, and the ability to specify what is meant by the security of a system.

This section is concerned with the meaning of system security and how it applies to our system. It presents

- some general situations that have permitted the security of existing systems to be violated,
- several typical user environments that should be supportable by a meaningfully secure system to overcome these violations,

- the meaning of security relative to such a system, and what it means to a user, and
- the role of proofs and the sense in which they are complete (e.g., under correct hardware operation).

In general, security is facilitated by a combination of factors, including the methodology itself, the design of the system and the design of user subsystems, their implementation, various administrative and other policies, and run-time monitoring.

With regard to the use and operation of systems that permit the flexible sharing of objects and the establishment of special security domains, there are numerous significant generic problems whose solutions are desired. Examples of these are as follows. (See, for example, Anderson and Edwards, quoted in Branstad [73].)

- Prevention of bypasses of the system protection mechanisms and the security policy, including the hardware enforcement mechanisms and software authorization (e.g., passwords and other identification or logic).
- Prevention of "Trojan horse" attacks upon the system, or upon unsuspecting users. These may involve the implantation of clandestine side effects in a compiler, in a system routine, or in a user-produced subsystem that gains acceptance and use by other users.
- Prevention of tampering with the live version of the system which could introduce arbitrary new possibilities for security violations.
- Prevention of improper design and/or implementation. Many types of potential violations arise from the visibility of implementation-dependent information that can implicitly or

explicitly give away other supposedly invisible information. Such examples may arise through timing idiosyncracies, through sequential dependences within an implementation, through incomplete parameter checking on calling or returning from a particular function, and through incomplete interrupt handling. Violations may also arise through incomplete clearing of residues, such as permitting newly allocated storage to be allocated before it is erased, or else to be read before it is overwritten.

These problems are all addressed by the methodology and by the design. To illustrate this, four particularly important security situations that can be attacked by the system are given below. These examples illustrate the properties to be proven about the system.

- Mediated access--sophisticated environments should be constructable, for example, permitting access authorization on a bit-by-bit basis within a data entity, depending on the user's identity, the time of day, what operation is being performed, and how often that operation or related operations have recently been performed. Such mediated access controls are easily handled by extended-type object managers.

- Mutual suspicion--each of two subjects (e.g., users or program environments) can permit controlled and revocable access by the other subject to only a specified subset of his objects. (Solutions permit the prevention of various forms of security bypass and Trojan horse attacks.) (See Schroeder [72].)

- Memoryless operation--one subject can allow another subject to operate on some of his objects in such a way that the other subject can neither retain nor transmit those objects or transform any of those objects. Attaining such an environment is known as the confinement problem (e.g., Lampson [73]).

- Military Security Classification and Need-to-Know--reading, appending and writing of classified documents are to be commensurate with established rules governing levels of subject clearance (top-secret, secret, etc.), levels of object classification, categories and "need-to-know." Access is to be nontamperable and nonbypassable (see Appendix C).

In each case, the methodology itself contributes to realizations for these situations. The specific design contributes still more, particularly the use of capabilities and extended-type objects (see Linden [74]). The methodology supports formal proofs of both invariant properties of what must not change, and variant properties about how the security state of the system is permitted to change. The nature of proof is that the system should do what is expected of it, and nothing else.

With respect to the operating system, there are two basic types of security violations, namely:

(T1) unauthorized modification of information, and

(T2) unauthorized acquisition of information.

The formal axiomatization of these principles is discussed in Chapter 8. Here "information" is used to mean programs, or data, or both. Programs implementing the operating system itself are of course included. Another type of violation beyond T1 and T2 involves

(T3) unauthorized denial of service.

Denial of service may arise from the unavilability of various classes of system resources, whether intentionally or unintentionally caused. Certain instances of denial of service are covered by T1 (i.e., unauthorized deletion), and need not be formalized independently of T1. A final type of violation involves

(T4) inference about apparently secure information.

The first line of defense against such violations is available in the stages of design and implementation. The next line of defense involves proofs. However, there are two particular cases in which such prevention is not always possible. One important case involves hardware errors, although detection of the most critical of these errors can be achieved by good (fault-tolerant) hardware design. The other case involves type T4, i.e., lack of confinement in a would-be memoryless environment, in which low-bandwidth signaling is achieved by statistical or deductive inference about information otherwise thought to be hidden (see Lampson [73]). Effects of the type T4 are intrinsically impossible to prevent completely, but can be reduced considerably by strict enforcement of the hierarchical structure and the doctrine of invisibility of implementation detail (including timing information). Finally, the third line of defense involves run-time monitoring of usage (see Chapter 9). Monitoring is of special interest in those areas where proofs are lacking or deficient, e.g., under circumstances involving potential errors due to hardware faults, and when denial of service is not explicitly prevented by the design. Monitoring is of course also of interest to security officers who wish to observe system usage.

The axiomatization of security assertions defining the nonoccurrence of T1 and T2 in terms of the O-functions and V-functions of the system and in terms of capabilities for objects is given in Chapter 8. This results in two corresponding principles that must be maintained:

P1: There shall be no unauthorized alteration of information (the Alteration Principle).

P2: There shall be no unauthorized acquisition of information (the Detection Principle).

Here "authorization" has meaning only with respect to a mapping from capabilities to the information in the system. Then the authorization to access a particular piece of information implies having access to an appropriate capability (itself obtained in an authorized way). Authorized passage of capabilities is strictly limited to two mechanisms, obtaining an object that contains a capability, or being passed a capability as a parameter of the call to or return from a procedure. (Note that the return from creating an object provides the capability for that object.) For both mechanisms, this passage is itself controlled by capabilities. As in other systems, a user's authorization thus ultimately depends on what is made available initially to the user upon logging into the system. (Additional constraints on authorization may of course also be enforced by the system, e.g., those of the multi-level security classification system.) The identification and initial authorization are thus critical to security, but are beyond the scope of the operating system design--apart from the fact that this initial authorization can itself be made secure. (See Chapter 6 for a further discussion of *initial authorization.*)

Formalization of the nonoccurrence of T3 is beyond the scope of the present effort, but analogous global assertions can be constructed for this case (denial of service), e.g., covering issues of termination (such as the fairness and noncompromisability of the scheduler) and prevention of deadlock. The issues are seemingly more difficult to state than for T1 and T2. Nonoccurrence of T4 is also beyond the present scope, although it appears to be partially amenable to formal treatment within the framework of the methodology and the system design.

The invariant security assertions P1 and P2 and corresponding variant assertions provide necessary conditions for security. However, security is in practice meaningful only with respect to specific use of the system. Thus for a given application (such as the multi-level classification system of Appendix C), it is desirable to define additional assertions relevant to that specific application. In each case, proofs of the satisfaction of these assertions rest on the existence of formal specifications for the appropriate level of the system.

We now consider the notion of security in the context of the specific system obtained, applying the methodology. With regard to each of the security assertions characterized by P1 and P2, the correctness of the security of the system relies on two considerations:

(C1) the correctness of the implementation of the model of interpreting O-, V- and OV-functions, e.g., with regard to indivisibility, invisibility, maintenance of the orderings among levels, and error handling; and

(C2) the correctness of the security assertions with respect to the specifications of each function (stage S2).

With regard to C1, we have hypothesized a model for the interpretation of V-, O-, and OV-functions that relates the two global security assertions to the possession of capabilities. This includes the following properties:

- Using a V-function cannot increase a user's intrinsic access rights of any object.
- Using an O-function cannot increase a user's access rights to any previously existing object.
- Using an OV-function or O-function to create a new object can increase a user's access rights; however, it provides access only to the newly created object, and not to any other object.

With regard to C2, the security assertions are appropriate for all functions (V, O, and OV) that are visible up to the user-visible operating system interface (level 10). As software to support specific applications is added, further assertions appropriate to each application must be added if proofs are desired for these applications. As an example, consider the multilevel security classification system mentioned above. In the MITRE work (Bell and LaPadula [74]), the following assertion is basic:

(P*) The content of a given object may be derived only from other objects whose classification level is at most that of the given object.

For example, a secret document may not include information taken out of a top-secret document. This assertion is called the "*-property" by Bell and LaPadula (see Appendix C).

For the system, eventual proofs of P1 and P2 (formally axiomatized) will require consideration of the hardware as well as the software. For example, the following steps contribute to proving that the basic protection mechanisms (capability creation in the hardware, and capability interpretation in hardware and software) are adequate:

- The generation of a capability is guaranteed to produce a new unique identifier.
- The only operation permitted on an existing capability is to reduce the access rights associated with it.
- The mechanism for accessing system-defined objects by means of capabilities cannot be bypassed or compromised.

Proofs of the security of a user application environment (e.g., the military security environment) are incremental to proofs of the security of the basic system. However, great care must be taken in stating appropriate security assertions. In the MITRE example above, it is not adequate to prove just the *-property, for nothing is said about the reclassification, declassification, or deletion of objects.

We intend to accomplish proofs manually, but we see the use of proof checkers (e.g., Boyer[74]) useful in the future in detecting errors in the proofs. Semiautomatic proving may also provide substantial help in the future. An on-line facility for interactive editing of specifications and their formal manipulation is considered vital to any future work.

1.5 Preliminary Conclusions

As a result of the work to date, the following conclusions are in order:

(1) The methodology has the potential to provide significant benefits to the design, implementation, integration, verification, monitoring, operation, maintenance, understandability, and evolution of an operating system designed according to it. It also permits the uniform treatment of a family of related systems, and enhances the portability of the design. These benefits result, among other things, from

- an integrated approach,
- use of formal specifications,
- use of hierarchical design structure and of hierarchical proofs,
- staging of design, implementation, and proof, and
- avoiding unnecessary hardware dependence in the design.

The methodology has already had impact on other work, and is being used in two other projects, one a MITRE project on redesigning the Multics kernel (for the Air Force), the other an SRI project on developing an ultrareliable computing system (hardware and software) for commercial aircraft (for NASA). Several other applications are currently being considered at SRI and at the University of Texas. Further work on the methodology is required in the area of concurrency and synchronization and in implementation.

(2) The use of capabilities presented here as the basis for the system design is of great help in

- structuring and unifying the design,
- providing solutions to hitherto difficult security problems, and
- proving the correct implementation of security.

In addition, capabilities are in general compatible with high-level authority-based designs.

(3) The design defined here has the following attributes:

- The system appears to be efficiently implementable with reasonable hardware constraints.
- The cost of building suitable hardware appears to be competitive with conventional designs.
- Security is attainable that can be more convincingly demonstrated than in existing systems.
- Problems whose solutions have appeared difficult in other systems can be solved efficiently.
- Proof of system security seems feasible with a reasonable amount of effort.
- System initialization, system alteration, recovery from faults and run-time monitoring of system security appear to be greatly simplified.

All in all it appears that the methodology has significant potential impact on the computer field, for both the short-term and long-term future. The secure operating system design has significant potential impact on new systems emerging in the future, as well as impact on existing efforts (e.g., at MIT and at MITRE).

Chapter 2

DESIGN ALTERNATIVES

In approaching the task of designing a secure operating system about whose design and implementation certain properties can be proven, it is first useful to examine various design alternatives and their suitability for attaining the goals set forth in the previous chapter. Three basic approaches are immediately evident, and are summarized as follows.

(1) Patching--Use an existing system, detect its deficiencies (possibly by penetration studies), and patch it.

(2) Language design--Design a language that can intrinsically enforce security via its own restrictiveness, or via compilation or interpretation, and implement a translator for it in a way that enforces its use for all users.

(3) System design--Design or redesign a system, and implement it.

Approach (3) comes in several varieties, with various degrees of sharing permitted, and various degrees of new design. The following examples are cited here:

(3a) Kernel redesign--Redesign the central part (kernel) of an existing system (MITRE; MIT-Multics modifications).

(3b) Virtual machine--Design the kernel of a system so that each user has an independent virtual machine. This is the "hypervisor" approach, and usually has highly restricted sharing of resources. An implementation of this kernel in separate hardware is called "encapsulation" (Lipner [74], Popek and Kline [74], Bisbey and Popek [74], Randell [75]).

(3c) Kernel design--Design a system kernel that handles the most primitive security functions (without the separate hardware restriction of 3b), but leaves much of the system undesigned (Hydra: Wulf et al. [74]; Burke [74]).

(3d) System design--Design a total system (Multics: Organick [72]; Plessey 250; CAP: Needham [72]; BCC 250).

Existing work on the design approaches is summarized in Section 2.1.

Early in the present work, approach (3d) was chosen. The reasons are already implicit in the preceding section and in Chapter 1, but are worth summarizing here.

- A general-purpose system is desired, with significant future impact, especially in the long term. (The two efforts to evolve Multics have near-future impact, namely the MITRE effort and the MIT-Honeywell effort. The MITRE-Multics effort is using a methodology similar to the work described here and has hopes of proving properties about a multilevel security classification system.)
- Specification of all visible system functions is necessary to make proofs of user-oriented security properties meaningful. For this reason, kernel approaches and partial designs were eschewed.
- A strongly hierarchical system is desired to facilitate proofs. Thus existing systems are mostly inappropriate.

2.1 Comparison of Various Approaches

A useful reference surveying various efforts to develop secure operating systems is given by (Saltzer [74]). Some of these efforts are resulting in systems in which sharing is not flexible and in which special security problems are not ‹fficiently solved, but which can be proved correct using current technology. There are two such approaches:

(1) The encapsulator (Lipner [74], Bisbey and Popek [74]) ensures totaı isolation by externally managed swapping of operating systems running on a given machine. Integrity of secondary storage is ensured by requiring the encapsulator to switch disc packs on-line, depending on which operating system is running.

(2) The virtual-machine monitor (Goldberg [74], Popek and Kline [74]) is a software system that provides a virtual machine (a copy of the hardware machine) for a user by allowing most hardware instructions in nonprivileged mode and by trapping all sensitive hardware instructions (such as I/O) to be simulated by the virtual-machine monitor. Popek and Kline have isolated a small portion of the virtual-machine monitor, called the kernel, the verification of which is sufficient to guarantee the security of the virtual-machine monitor.

However, these methods provide only a small subset of the services that would be expected of an operating system. For example, they do not provide facilities for sharing or for the solution of the special security problems. Still, they represent viable short-term solutions to verifiable security in some form.

The Multics system (Organick [72]) represents the best existing system that satisfies many of the properties desired here. Relative to other systems, Multics contains few loopholes for penetration. However, Multics, in its present form, does not permit the efficient solution of the special security problems mentioned in Chapter 1; and its sheer size and lack of structure preclude any possibility of proof.

There are two current projects--one at MIT and the other at MITRE--aimed at isolating the functions of Multics relevant to security (see Saltzer [74]). The hope is that the residue (a *security kernel*) will be small enough to verify by inspection (the MIT approach), or to prove formally (the MITRE approach). Another security kernel effort is the Hydra system (Wulf et al. [74]), which appears to have sufficient mechanism to solve a wide variety of special security problems but is not appropriate for verification, due to the lack of formal specifications, formal assertions, and suitable structure. Another effort exists at MITRE (Burke [74]), aimed at proving a security kernel for the PDP-11.

A security kernel, in addition to arbitrating all references to resources (which it must do to guarantee isolation of users), also provides primitives for the use of the normal operating system features that were pruned off to form the kernel. Such eliminated features include subsystems that theoretically affect only system performance, e.g., page-replacement programs and process scheduling programs. However, some aspects of these programs could result in damaging effects, since they stand between a user (or set of users) and the kernel. In particular, efforts in non-kernel operating system programs could result in:

- Denial of service--With unfair algorithms, certain users could be excluded from their share of computing time.
- Leakage of information (which Lampson [73] calls lack of confinement)--Here these programs have knowledge of many of the operations performed by users (e.g., file opening, requests for I/O), and can act as information channels between users.

The present approach is to produce a complete operating system for which formal specifications are provided for all functions that a user can call. Thus it is theoretically possible that any security-related property of the system could be derived. In addition, it should be possible to state and prove properties of the programs that implement the special security subsystems. As noted in Chapter 1, however, denial of service seems amenable to formal treatment, while leakage of information is only partially amenable to such treatment.

In scope, the present work is more similar to Multics than any of the other work mentioned above. Nevertheless, it is significantly different from Multics, with respect to the use of formal specifications, the use of an explicit hierarchical structure of the operating system, the use of capabilities rather than descriptors, and with respect to having been begun with both proof and formally definable security in mind from the beginning. (Further, the work has taken significant advantage of the Multics experience.) As far as the use of capabilities is concerned, it appears that no complete design for a general-purpose capability-based operating system has previous been developed.

The methodology bears a resemblance in its goals to the LOGOS project (Glaser et al. [72]), which is also aimed at systematizing the design process, modeling a system (or at least parts of a system), and demonstrating consistency. However, LOGOS deals primarily with graphical representations for individual pieces of a system, e.g., focusing on determinancy and termination properties of particular algorithms. It lacks sufficiently formal abstract representations to handle the entire system at different levels.

As noted in Chapter 1, the methodology presented here greatly reduces the effort in realizing a total system. The system emerging therefrom appears to be suitably general, demonstrably secure, and realistically and efficiently implementable. It is of course possible to use the user-visible operating system interface (i.e., level 10) in many application areas, creating totally distinct command interfaces.

Perhaps even more significant in the long run, the methodology itself appears to be readily transportable to manufacturers, software houses, and other developers of hardware and software. The system design also appears to be transportable in a sense not hitherto available, e.g., by using as many of the higher levels as desired down to an interface appropriate to the hardware in question.

At this point, the reader interested in a formal exposition of the methodology should proceed to Chapter 3. The reader not so inclined, but interested in the design of the operating system, may wish to skim over Chapter 3, and then move on to Chapter 4 and following chapters. A reader seeking an intermediate level of detail may wish to read Robinson et al. [75].

Chapter 3

THE METHODOLOGY

The methodology as described in Chapter 1 is an extension of the techniques presented in Robinson and Levitt [75]. As stated earlier, the hierarchical methodology for design, implementation, and proof utilizes

(1) decomposition of the system into a hierarchy of abstract machines,

(2) formal specifications for each abstract machine as a Parnas module,

(3) assertions concerning representations of the status of each abstract machine in terms of the states of machines at lower levels,

(4) abstract programs using lower-level functions, and

(5) realization of the most primitive machine and the abstract machine operations in terms of hardware or a programming language.

In this chapter, the use of Parnas modules and hierarchical decomposition is motivated. The stages of development are then discussed in detail. Finally, a simple example is given.

In recent years there has been increased recognition of the benefits of using programming methods that introduce several refinements (i.e., layers of abstraction) between the statement of the programming task and the ultimate runnable code. The stepwise refinement method (Dijkstra [72]) allows a programmer to elaborate his concept of a program in successively finer detail and to make convincing arguments of the program's correctness.

Our methodology is based on a formalization of the refinement concept. We hypothesize a sequence of abstract machines (M_0, M_1, ..., M_n), with M_0 being the most primitive layer (possibly the hardware or a programming language). Viewed externally, each abstract machine contains various functions that a user of the machine can call (either returning data structure values or performing operations that modify the data structures). If M_0 is realized as hardware, the data structures are the register set and memory, and the operations are the instruction set. If M_0 is represented by a programming language, the data structures are the program-variables, and the operations are the assignment and control mechanisms.

Consider the task of writing a program P that is to run on M_0. In the hierarchical methodology, we write a program P_n that will run on M_n. Then we write, for each i, $0 \leq i < n$, a sequence of programs $\{P\}_i$ that implement M_{i+1} and run on M_i. Each of the pairs ($\{P\}_i$, M_i) constitutes an abstraction, or equivalently, a level.

It is clear that in a hierarchy, an abstract machine should not be able to call functions of machines at a higher level. However, there are two possible views concerning the calling of lower-level functions.

In the first view, M_i can potentially call functions of all machines below it. In this view, in order to preserve a hierarchy, M_i (for $1 < i \leq n$) may call functions of M_j, $0 \leq j < i-1$, only if M_{i-1} can call them as well. In the other view, a machine M_i' is able to call only fuctions of M_{i-1}' (the machine just below M_i'). We prefer to describe a hierarchical system according to the first view, because it simplifies the formal description of the machines, and because it actually reflects the implementation of the operating system. On the other hand, we use the second view to prove properties of the system. There is a construction, shown in Figure 3.1, to convert a hierarchy of the first view (Figure 3.1a) into an equivalent one of the second view (Figure 3.1b). In this construction, the machine M_i' at level i consists of the union of (up to)

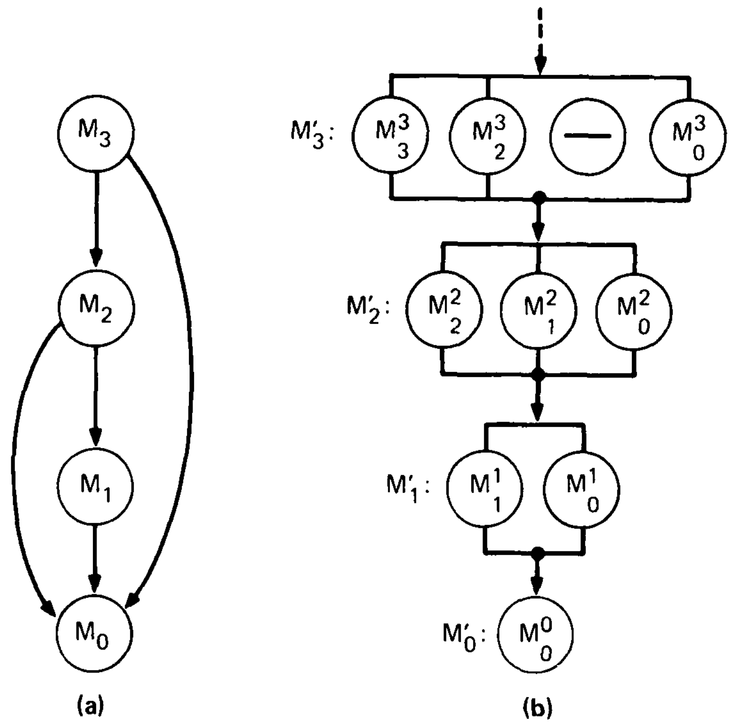


FIGURE 3.1 TWO VIEWS OF A HIERARCHY WITH MULTI-LEVEL VISIBILITY

i machines M^i_j, $0 \leq j \leq i$, where each M^i_j contains precisely those functions of M_j (in the first view) that are visible at level i. (Note that M^i_i is thus identical to M_i of the first view.) Also note that M^i_j, $0 < j < i$, contains only a subset of the visible functions of M^{i-1}_j.

The hierarchical methodology requires a medium for formally describing each of the abstract machines. To achieve this, we represent each machine as a Parnas module (Parnas [72a,b]) that can be viewed as a generalization of a finite-state machine. A module has a state, and some externally visible operations to change the state. The state of the machine can be derived by calling a set of V-functions (value-returning), which can have arguments over a potentially infinite domain and can have a potentially infinite range. The state of the module is changed by calling a member of the set of O-functions (*o*peration), which can also have arguments over a potentially infinite domain.

The semantics of a module is described by a formal specification that shows the result of either an O-function or a V-function call.

For a V-function, the specification includes: the type of the V-function value--integer, boolean, etc.; a listing of the parameters and their types; the initial value of the V-function; and a list of exception conditions. The value of the V-function for particular arguments could be the special value UNDEFINED. An *exception condition* is an expression which, if satisfied at the time of a V-function call, implies that no value will be returned. Instead, an appropriate error routine (supplied by the caller and executing in the caller's domain) is invoked. Parnas envisaged several reasons for associating exception conditions with a call on a V-function, one of which is to ensure that no call is allowed such that the value UNDEFINED would be returned. That is, a V-function should never return the value UNDEFINED, but the user should be alerted, by an invocation of an error routine, if he calls a V-function out of range. In the operating system, an exception condition is also triggered when a protection violation would occur if the V-function were allowed to return a value.

For an O-function, the specification includes: the parameters and their respective types; the exception conditions; and the *effects* section. Here, Parnas envisaged exception conditions as expressions defining O-function argument values whose use would be improper, e.g., to cause the range of a V-function to be exceeded. An O-function call that triggers an exception results in no state change to the module. As in the case of V-function exceptions, the satisfaction of an exception condition causes the invocation of an error routine. We also use an exception condition to detect a protection violation, i.e., a call wherein the caller does not have sufficient rights. The "effects" section defines the state change due to the O-function call by giving the new values of V-functions in terms of V-function values prior to the call. Both the effects and the exception conditions are described by *assertions*, written in terms of the V-functions of the module and a base assertion language that can be formally defined.

The O- and V-functions provide the interface that the module presents to a caller. It is possible that certain V-functions are not to be made available at the interface. We call these *hidden V-functions*. They have a specification analogous to ordinary V-functions (since they are a part of the module state), but do not possess exception conditions since there is no means for a user of a module to call them.

Another special type of V-function is the *derived V-function*. Its specification possesses exception conditions, but its value is expressed in terms of the values of nonderived V-functions (possibly including hidden V-functions). A derived V-function never appears in the effects section of an O-function, since its value is always derivable from other V-functions. In our operating system specifications, most of the user-visible V-functions are derived (with the appropriate exception conditions).

One additional special function is an *OV-function*, which returns a value (like a V-function) and performs a state change (like an O-function). An OV function is needed as an indivisible operation in a multiprogramming environment where a module is shared among several users. Here a state change is to be effected, and some values depending on the new state are to be returned to the caller--before another user can call an O-function that will change the state again. The use of OV-functions provides a convenient abstraction that would necessitate a more complex specification if not allowed. (In the system specified in Appendix A, user-visible OV-functions are used only for the creation of capabilities.)

Each function is indivisible to the caller. During an O-function call, the V-function values associated with the call form a "critical section" that excludes V-function calls or other O-function calls affecting the critical V-function values.

We have outlined above how we intend to decompose a system into a hierarchical layering of abstract machines (and implementations of abstract machines) similar to the method of Dijkstra [68c]. We have also provided an additional degree of decomposition to separate the tasks of specification, design, and implementation. There is also a separate proof effort associated with each task. In particular, the methodology involves five stages as follows.

Stage 1--In this stage the generalized facilities visible to the user of the software system are first determined. Then several steps occur, not necessarily in order: the set of facilities are grouped into modules; the functions (V, O, and OV) associated with each module are determined; and the modules are arranged in a hierarchical ordering. This is a very critical stage (especially the ordering of modules), because at this point the general configuration of the system has been determined.

In our system we first decided that we wanted capabilities, virtual memory, extended types, directories, process management, and dynamic linking. We then determined the approximate relationship among machines in the hierarchy, and finally wrote out the functions. The most important decision was to put capabilities and memory mapping at the lowest system levels. The virtual memory is in effect "divided" into several levels, as is process management (see Chapter 5). Other examples of difficult design decisions at this stage are:

(1) the placement of revocation at the same level as segment management.

(2) the placement of user processes immediately below command level.

(3) the placement of extended types immediately above segments, enabling the use of the extended type mechanism to define operating system structures, and

(4) the creation of a separate level to solve the lost-object problem, together with a careful choice of functions at levels 4, 5, 6, and 7 to work out a consistent solution.

Also at this stage is the formulation of global assertions for each level. These assertions represent general properties that the specifications at a given level must adhere to. An example of a global assertion is the lost-object assertion: for every object above level 7, there must be a distinguished directory entry corresponding to it. For the present work, the global assertions of interest are those relating to security (see Chapter 8).

Stage 2--Each module is formally specified, according to our extension of Parnas' methodology described above. Based on the specifications for a module, it is possible to attempt to prove that the specifications are self-consistent and that certain global assertions are true. A proof of self-consistency involves demonstrating that no set of assertions in the effects section is self-contradictory over the domain of definition for the V-functions. In general, this proof is done almost by inspection, In general, an inconsistent module specification prevents a proof of the implementation of the module from being successful, and thus a proof of inconsistency need not be explicitly carried out. However, for diagnostic purposes, it is useful to do consistency checking at this stage. Global assertions (see Price [73]) are expressions written in terms of the

V-functions of the module. To show that they apply to the module, it must be shown that they are true for the initial state of the module and also after any sequence of O-function calls. Global assertions describe general properties of a module, and thus may be used as lemmas to simplify proofs of abstract programs that call functions of the module. In Stage 2 we state and prove potentially useful global assertions for each of the modules with respect to the specifications. General system properties (e.g., those pertaining to security) can be represented as global assertions. General system properties are associated with a module called the user interface. In a hierarchical system as described by the view of Figure 3.1a, the user interface corresponds to machine M_n' in Figure 3.1b, containing functions of machines initially defined at many levels.

An example of a module specification is shown in Table 3.1. The module (called a Register Module) maintains a register and was originally used by Parnas ([72a]) to execute Markov algorithms.

A module specification has four parts: PARAMETERS, DEFINITIONS, EXCEPTIONS, and FUNCTIONS. The PARAMETERS section contains the type information for variables used in the module, as well as module resource limits that have been parameterized (none in this example). DEFINITIONS are module-wide macros (none in this example). EXCEPTIONS are the macro-definitions for the named exception condition. The FUNCTIONS section contains the specifications for the functions of the module. Using the Register Module, one can find the number of elements in the register (length) or the value of a particular element in the register (char), and one can insert (insert) an element into the register or delete (delete) an element from the register.

Both the V-functions and the parameters have types--either integer, boolean, or character in this case. The purpose is a description of the function in natural language. Each V-function has an initial value, which is described by an expression. Calls to both V-functions and O-functions of a module are restricted by *exception conditions*, which represent calls that cannot be handled by the module. For example, the call char(length+1) would yield an undefined value if allowed to proceed, and the call insert(i,j) with a length ≥ 1000 would--if permitted--overflow the register. We can often prove that a calling program never triggers an exception condition, as we do in the example of this paper, and thus preclude the necessity to perform a run-time check for exception conditions. However, it is often useful to construct programs that do trigger exception conditions (Parnas [72d]). In that case, the exception-handling program--and the accompanying transfer of control--must be considered in the verification.

The state transformations are described in the EFFECTS section of an O-function. The initial values, exception conditions, and effects are all written as assertions. V-function names within single quotes ('...') represent values of V-functions prior to the O-function call, while unquoted names represent new function values at the completion of the call.

Several examples of global assertions for the Register Module are:

0 ≤ LENGTH ≤ 1000;
∀i (0 < i ≤ LENGTH ⊃ *defined*(CHAR(i)));
∀i (defined(CHAR(i)) ⊃ 0 ≤ CHAR(i) ≤ 255);
defined(LENGTH).

(The predicate *defined* means that a V-function has a value not equal to the distinguished value *undefined*.) These invariants can be used as lemmas to shorten the proof of programs that call the module.

Stage 3--In this stage, decisions are made regarding the representations (or mappings) of the data structures of level i (characterized by the V-functions of level i) in terms of data structures (V-functions) of lower levels. We can also state and prove properties of the representations. This work is related to that of Hoare [72]. For simplicity we discuss in detail the case where the data structures of each machine M_i' are represented solely in terms of the data structures of the single lower-level machine M_{i-1}'. The generalization to the hierarchy of Figure 3.1a is straightforward.

The set of possible V-function values for an abstract machine can be viewed as the state space for that abstract machine. In terms of the state space S of M_i and the state space T of M_{i-1}, we define a <u>mapping function</u> of level i as an "onto" partial function from T to S. We first show that this concept of a mapping function conforms to the desired properties of the data representation. We then give a method for writing relations among V-function values of M_i and those of M_{i-1} in a manner that ensures that the corresponding state functions constitute a mapping function.

With regard to the state mappings from T to S, we observe the following:

- Numerous states in M_{i-1} can map to a single state in M_i, as depicted in Figure 3.2a, due to the possibility of delaying the decision on the precise representation of a data structure of M_i. In Appendix B we illustrate a representation for directories in terms of segments, in which the segment displacement that corresponds to a particular entry is not bound (i.e., the specifications and the mapping functions describe directory entries in an order-independent way). Of course, when all implementation decisions are made, the nondeterminism in the mapping disappears.

- Not all states of M_{i-1} have images in M_i. As illustrated in Figure 3.2b, a direct transition (S_1,S_2) in M_i might correspond to a transition (T_1,T_2) in M_{i-1} that traverses several intermediate states that need not necessarily map to states

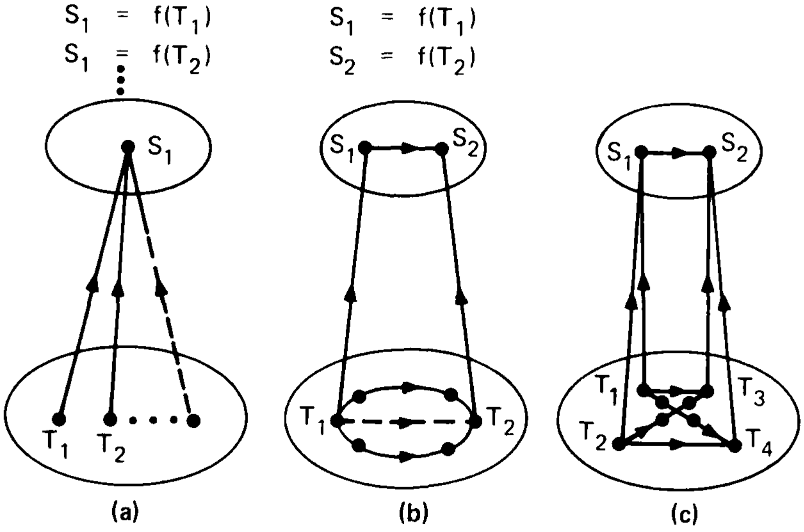


FIGURE 3.2 MAPPING FUNCTION f RELATING THE STATES OF TWO ABSTRACT MACHINES

in M_i. Moreover, there might be numerous possible paths between T_1 and T_2 (as shown) corresponding to different implementation algorithms.

- The most general case is that in which each of the states in a direct :ransition pair of M_i corresponds to several states in M_{i-1}. In Figure 3.2c, S_1 corresponds to T_1 and T_2, and S_2 corresponds to T_3 and T_4. A correct implementation of the direct transition (S_1,S_2) in M_i could be any of the following transitions in M_{i-1}: (T_1,T_3), (T_1,T_4), (T_2,T_3), or (T_2,T_4). Note that T_1 may be equal to T_2, or T_3 may be equal to T_4.

We write a mapping function for level i as a set of expressions containing the V-function values of machines M_i and M_{i-1}. Each expression, called a mapping function expression, is of the form $\langle V_i \rangle$: $\langle$expression$\rangle$ where V_i is a V-function value of M_i (with formal parameters), and "$\langle$expression$\rangle$" is written in terms of V-function values of M_{i-1}. A mapping function expression states the value of the higher-level V-function in terms of the lower-level V-function values.

Once mapping function expressions have been defined for each of the V-functions of M_i, it remains to be determined if they characterize the properties of a mapping function, as enumerated in the state-space description of abstract machines. We call the mapping function expressions consistent if such is the case. If mapping function expressions are inconsistent, it is impossible to find an implementation satisfying the specifications of the modules M_i and M_{i-1}. We prove the consistency of mapping function expressions between M_i and M_{i-1}, with respect to the specifications of M_i by creating mapped specifications for M_i, i.e., substituting each V-function reference in the specification of M_i by its instantiated mapping function expression. The mapped specifications can be proved consistent or inspected to check consistency, in the same manner used to show the consistency of a module specification in stage 2.

As we discuss below, mapping function expressions are used to transform module specifications of M_i into assertions expressed in terms of only V-functions of M_{i-1}. Also, a mapping function expression for a V-function of M_i serves as the output assertion for the program that implements the V-function.

We describe an implementation of a register in terms of an Array Module, described in Table 3.2. The functions of the array module permit the retrieval of the value of a position in a one-dimensional integer array of 1000 elements ("access"), and the changing of the value of an array position ("change"). The mapping function expressions for the Register Module are as follows:

length: ℓ (ℓ is a program variable)
char(i): IF $1 \le i \le \ell$ THEN access(i)
ELSE UNDEFINED

Mapped specifications for the Register Module are shown in Table 3.3. The mapped specifications, when simplified, yield easily readable assertions for the implementing programs. For example, the simplified output assertion for "insert(i,c)" is

$$\forall j(1 \le j \le i)[\text{access}(j) = \text{'access'}(j)] \wedge$$
$$\text{access}(i+1) = c \wedge$$
$$\forall j(i+2 \le j \le \text{'}\ell\text{'}+1)[\text{access}(j) = \text{'access'}(j,1)]$$

The effort in the first three stages results in what we call a design for a system. In our notion of a design, many of the important system properties can be stated and proved before any code is written. The specification of M_i and mapping function of level i are sufficient to generate the correctness criteria (i.e., input and output assertions) for the implementation of M_i.

Stage 4--Each of the functions of M_i, $i > 0$, is implemented as an abstract program using the functions of M_{i-1} (as in Stage 3, levels lower than i-1 are omitted for ease of description), and the control constructs of some formally-defined programming language. These programs complete the binding of the decisions that were left incomplete by the mapping functions of Stage 3. Each of these abstract programs must be proven to be a "successful" implementation, with respect to the specifications of M_i and to the mapping functions between M_i and M_{i-1}. We accomplish this by deriving input and output assertions for the implementing programs, which (if satisfied) imply such a "successful" implementation. The input and output assertions for the implementing programs are simply the mapped specifications of M_i. Then we prove the correctness of the implementing programs with respect to these assertions using an extension of Floyd's method described below. The extension defines correctness criteria for programs calling O-functions.

The problem of verifying (or proving the correctness of) an abstract program is similar to that of verifying any program, except that calls to O- and V-functions must be included in the semantics of the programming language and of the verification process. We will illustrate the verification of an abstract program, with emphasis on how a semi-automatic verification system might find a proof.

Floyd's method (Floyd [67]) is used in most automated verification systems (e.g., Elspas et al. [73]), and is the basis for the following discussions. The goal is to prove the correctness of a program, P, with respect to an input assertion, φ, and an output assertion, ψ. Verification requires the insertion of inductive assertions, $\{q_i\}$, into the program's flowchart, breaking the program into *simple paths*. Each simple path has one entry and one exit, and between these a fixed number of executable statements. For each simple path, a formula called a *verification condition* (VC) must be stated, and proved to be a theorem. The validity of all the VCs for a program is sufficient to demonstrate the *partial correctness* of a program--i.e., for all inputs satisfying the input assertion, the output assertion is satisfied if the program terminates. Termination can be proven by inductive assertions (usually different from those used to prove partial correctness) that bound the number of loop executions (Manna and Pnueli [74] and Dijkstra [74]).

VCs can be generated mechanically by a part of the verification system called the *verification condition generator*, which contains knowledge about the programming language and the assertion language, and takes as input the code for a simple path and the bracketing assertions. Proof of the VCs is attempted by a part of the verification system called the *deductive system* (i.e., a theorem prover for formulas in the assertion language).

In Floyd's method, each VC must be built from the antecedent assertion, q_i, which applies to the beginning of a simple path, and the consequent assertion, q_j, which applies to the end of the path. The goal is to arrive at a formula $\hat{q}_i \supset \hat{q}_j$ where $\hat{q}_i$ and $\hat{q}_j$ are the results of transformations reflecting the effects of intervening program steps. The n^{th} transformation ($n \geq 0$) takes the form $\{q_i^n \rightarrow q_i^{n+1}, q_j^n \rightarrow q_j^{n+1}\}$. After the last such transformation, we set $\hat{q}_i = q_i^{final}$ and $\hat{q}_j = q_j^{final}$, defining the verification condition. For simplicity, we group the transformations handled by current verifiers into three types corresponding to those programming language constructs.

- In assignment, the code associates new values with program variables, so that q_i and q_j may be referring to the same variable name but to different values. The updating of values is reflected in the consequent assertion by substitution:

$$x \leftarrow e \Rightarrow q_j^{n+1} = \{q_j^n\}_e^x$$

where $\{\alpha\}_e^x$ indicates replacement in α of all occurrences of the variable x with the expression e. Two different methods of substitution are used, differing in the order in which the code is scanned but yielding the same VCs: backward substitution (King [69], Igarashi et al. [73], Elspas et al. [73], and forward substitution (Deutsch [73]).

- The results of a test yield terms A that are conjoined to the antecedent assertion in a manner depending on the direction of the branch:

$$\text{test: } A \ (\underline{\text{true}}) \Rightarrow q_i^{n+1} = q_i^n \wedge A$$

$$\text{test: } A \ (\underline{\text{false}}) \Rightarrow q_i^{n+1} = q_i^n \wedge \neg A$$

- *Procedure calls* (Hoare [71]) yield transformations in both antecedent and consequent assertions:

$$\underline{\text{call}}\ p(a_i,\ldots,a_m) = q_i^{n+1} \Rightarrow q_i^{n} \wedge \{\psi_p\}^{f_1,\ldots,f_m}_{a_1,\ldots,a_m}$$

$$q_j^{n+1} \Rightarrow q_j \wedge \{\varphi_p\}^{f_1,\ldots,f_m}_{a_1,\ldots,a_m}$$

  where φ_p and ψ_p are the input and output assertions, respectively, of a procedure p. $f_1,\ldots,f_m$ are the *formal parameters* of the procedure, for which the actual parameters $a_1,\ldots,a_m$ of the call must be substituted in φ_p and ψ_p (the notation above indicates multiple substitution). This scheme works for call-by-value parameter passing, but does not work for call-by-reference unless there are restrictions on the choice of actual parameters (Hoare [71]).

We now describe the semantics of an O-function call in the generation of verification conditions. An O-function can be described as an *assertion pair*, with input assertion φ_O corresponding to the complement of the exception conditions for the O-function, and output assertion ψ_O corresponding to the effects of the O-function. An O-function call changes the values returned by V-functions. In the specification of an O-function, V-function values before the call are enclosed in single quotation marks, and V-function values after the call are unquoted. Thus, after each O-function call, a completely different set of V-function values comes into existence, and must be distinguished from the old set when verification conditions are generated. The problem is complicated by the fact that a simple path may have more than one O-function call: each O-function introduces a new set of V-function values into the universe of discourse. Our solution is to have the last set of V-function values (i.e., those at the point of the consequent assertion) be unquoted and the n^{th} set of V-function values before that be surrounded by n levels of single quotation marks. For example, if the antecedent assertion for a path is $\forall x(1 \le x \le 10)\ [F(x) = x]$,[*] followed by the execution of two O-functions--first to change the value of F(2) to 3 and then to change the value of F(3) to 2, the antecedent assertion would be modified to look like this:

$$\begin{aligned}
&\forall x(1 \le x \le 10)[\text{"F"}(x) = x] \wedge \\
&\forall x(1 \le x \le 10)[\text{'F'}(x) = \underline{\text{if}}\ x = 2\ \underline{\text{then}}\ 3 \\
&\qquad\qquad \underline{\text{else}}\ \text{"F"}(x)] \wedge \\
&\forall x(1 \le x \le 10)[F(x) = \underline{\text{if}}\ x = 3\ \underline{\text{then}}\ 2 \\
&\qquad\qquad \text{else 'F]}(x)] \quad .
\end{aligned}$$

(Initially the antecedent and consequent assertions look like this:

$$\begin{aligned}
q_i^{\text{initial}} &= q_i \\
q_j^{\text{initial}} &= \underline{\text{true}} \quad .)
\end{aligned}$$

The effects of each O-function call in the path are applied one at a time as the path is scanned in the forward direction. The antecedent assertion above can be simplified to the following:

$$\begin{aligned}
&\forall x(1 \le x \le 10)[F(x) = \underline{\text{if}}\ x = 2\ \underline{\text{then}}\ 3 \\
&\qquad\qquad \underline{\text{else}}\ \underline{\text{if}}\ x = 3\ \underline{\text{then}}\ 2 \\
&\qquad\qquad \underline{\text{else}}\ x] \quad .
\end{aligned}$$

For describing the general effect of increasing the level of quotation in an assertion, we use the function BUMPQUOTE(q) to indicate an increase of 1 in the quotation nesting levels for every V-function call of assertion q. For an O-function call $O(a_1,\ldots,a_m)$, where the O-function input

[*] This means for all values of x such that $1 \le x \le 10$, then $F(x) = x$.

assertion φ_0 and output assertion ψ_0, the assertion transformations are as follows:

$$q_i^{n+1} = \text{BUMPQUOTE}(q_i^n) \wedge \{\psi_0\}_{a_1,\dots,a_m}^{f_1,\dots,f_m}$$

$$q_j^{n+1} = \text{BUMPQUOTE}(q_j^n) \wedge \{\varphi_0\}_{a_1,\dots,a_m}^{f_1,\dots,f_m} \quad .$$

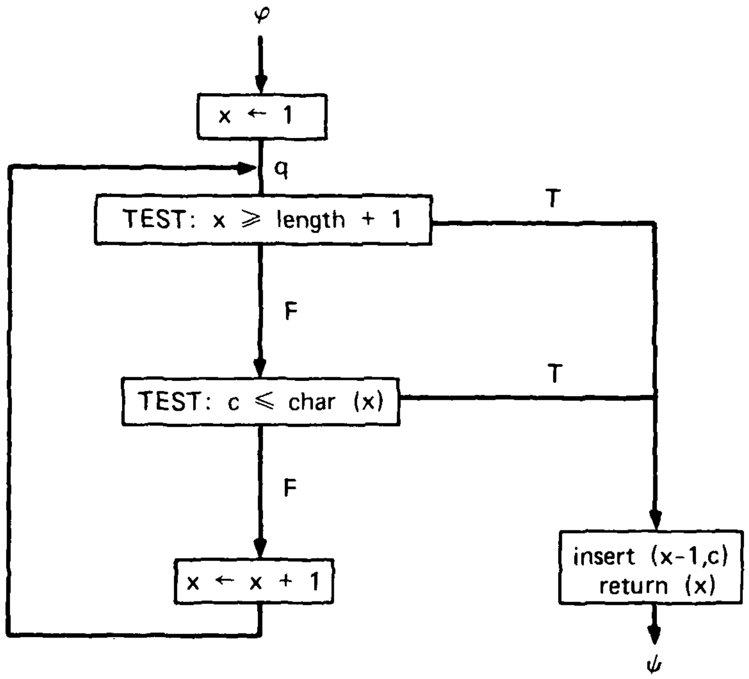
This formulation of verification conditions assumes that an O-function call will never trigger an exception condition. In fact, this is guaranteed by the conjunction of φ_0 to the consequent assertion. This assumes that $\{\varphi_0\}_{a_1,\dots,a_m}^{f_1,\dots,f_m}$ has all V-function calls in a single level of quotes. The maximum level of quoting refers to V-function values prior to the first O-function call, the next largest level to V-function values prior to the second O-function call, and so on. After the last O-function in the path is handled (when the assertions are q_i^n and q_j^n), the final transformation is as follows:

$$\hat{q} = q_i^n$$

$$\hat{a} = q_j^n \wedge q_j \quad .$$

If the assertions in the effects section of an O-function define a unique assignment of V-function values, then the simplification performed above in the example can be made immediately. This substitution, which resembles the semantics of an assignment statement, is called functional assignment.

In the next example, we prove the correctness of INSERTSORTED, a program that inserts a character c (denoted by an integer code) into a previously sorted register, while maintaining its sorted state, and returns the index of the inserted character. The program uses the Register Module described in the previous section. The flowchart and assertions for INSERTSORTED are shown in Figure 3.3. The input assertion, φ, expresses



φ: lpred (1, (length-1), k, (char(k)
    $\leqslant$ char (k+1)))

ψ: length = 'length' + 1 $\wedge$
lpred (1, (length-1), k, (char(k)
    $\leqslant$ char(k+1))) $\wedge$
lpred(1, (length-1), k, (<u>if</u> 'char'(k) < c <u>then</u>
    char(k) = 'char'(k)
    <u>else</u> char(k+1) = 'char'(k)))$\wedge$
char(x) = c

q: length = 'length' $\wedge$
lpred (1, length, k, (char(k) = 'char'(k))) $\wedge$
lpred (1,(length-1), k, (char(k) $\leqslant$ char(k+1)))$\wedge$
lpred (1, (x-1), k, (char(k) < c))$\wedge$
x $\leqslant$ length + 1

SA-2581-8

FIGURE 3.3 FLOWCHART AND ASSERTIONS OF PROGRAM "INSERTSORTED"

the fact that the register is initially sorted. The function LPRED (x, y, k, rel(k)), called the linear predicate, means that the sequence of predicates rel(x), rel(x+1),..., rel(y) are all true,* where rel(k) is a formula containing a free variable k. This special case of universal quantification is easier to process automatically, because special rules of inference can be used that would not apply to the general case of universally quantified expressions. Two examples of such rules are:

$$x > y \supset \text{LPRED}\ (x,\ y,\ k,\ rel(k))$$

and

$$\text{LPRED}\ (x,\ y,\ k,\ rel(k)) \wedge rel(y + 1) \supset$$
$$\text{LPRED}(x,\ (y + 1),\ k,\ rel(k)).$$

The output assertion, ψ, expresses the following conditions: (1) the length of the register has been incremented, (2) the register is sorted, (3) all values of the original register are in the final register--the conservation condition, and (4) the given character was actually inserted. Since only one path contains an O-function call, a V-function name in single quotes ('...') within any assertion refers to the value of the V-function at program entry.

Insertion of the inductive assertion, q, enables a description of the flowchart as a set of simple paths. To illustrate the proof technique, we describe the verification of a path from q to ψ through the "false" exit of the first test, through the "true" exit of the second test. We must show that no function call along the path satisfies an exception condition, which follows from (x < length + 1)--the results of the first test. The verification condition for this path is derived by substitution into the consequent assertion of the effects of the call insert(x-1, c), and by conjoining the effects of the two tests to the antecedent assertion of

* $\forall k(x < k \leq y)[rel(k)]$.

the path. The following formula shows only the part of the substituted consequent assertion corresponding to the second conjunct of ψ:

$$\begin{array}{llll}
\{\mathrm{LPRED}\ (1,\ \mathrm{length}-1),\ k,\ (\mathrm{char}(k) \le \mathrm{char}(k+1))) \wedge & & & (1)\\
\mathrm{LPRED}\ (1,\ (x-1),\ k,\ (\mathrm{char}(k) < c)) \wedge & & & (2)\\
(x \le \mathrm{length}+1) \wedge & & & (3)\\
(c \le \mathrm{char}(x)) \wedge & & & (4)\\
(\neg\ (x \ge \mathrm{length}+1))\} \supset & & & (5)\\
\mathrm{LPRED}\ (1,\ \mathrm{length},\ k, & \{(\underline{\mathrm{if}}\ k \le x-1\ \underline{\mathrm{then}}\ \mathrm{char}(k) & (A) & (6)\\
 & \underline{\mathrm{else}}\ \underline{\mathrm{if}}\ k = x-1+1\ \underline{\mathrm{then}}\ c & (B) & \\
 & \underline{\mathrm{else}}\ \mathrm{char}(k-1)) \le & (C) & \\
 & (\underline{\mathrm{if}}\ k+1 \le x-1\ \underline{\mathrm{then}}\ \mathrm{char}(k+1) & (i) & \\
 & \underline{\mathrm{else}}\ \underline{\mathrm{if}}\ k+1 = x-1+1\ \underline{\mathrm{then}}\ c & (ii) & \\
 & \underline{\mathrm{else}}\ \mathrm{char}(k+1-1))\}) & (iii) &
\end{array}$$

Antecedent conjuncts (1)-(3) are from q; (4)-(5) are the results of the two tests; (6) comes from the second conjunct of ψ.

Such a formula can be proven by an automatic theorem proving program. An example of such a program is the deductive system of the SRI verifier. We use rules similar to those of this system to present a sample proof here.

The SRI verifier, which uses a natural-induction approach to theorem proving, is implemented in the high-level, problem-solving language QLISP (Reboh and Sacerdoti [73]), a successor to QA4 (Rulifson et al. [72]). For more details on the deductive system of the SRI verifier, see Waldinger and Levitt [73]. This deductive system works primarily in a goal-directed manner. The consequent of the VC becomes the *goal*, and the conjuncts of the antecedent of the VC are put into the *data base* (i.e., facts the system knows). A step is proven by performing operations on the goal (such as substitution of data-base items, simplification, or breakdown into subgoals), corresponding to heuristics or rules of inference. The proof is complete when all the derived goals are equivalent to items in the data base. Of course, a heuristic search of proof sequences must be made in order to find a valid proof. An example of a simple but effective inference rule is: If it is required to prove f(a) = f(b), first try to prove a = b.

The verifier would try to prove goal (6) by breaking it up into nine cases: the combinations of the three cases of each of its two conditional expressions, A, B, C, and i, ii, iii. It would make use of the context mechanism in QLISP, a feature that enables incremental changes to the data base depending on some past context. The deductive system can build a tree of contexts in which the "if" relations are asserted to be appropriately <u>true</u> or <u>false</u>. For example, in one context we can assert

$$k \leq x\text{-}1 \qquad (7)$$

and

$$k+1 \leq x\text{-}1 \qquad (8)$$

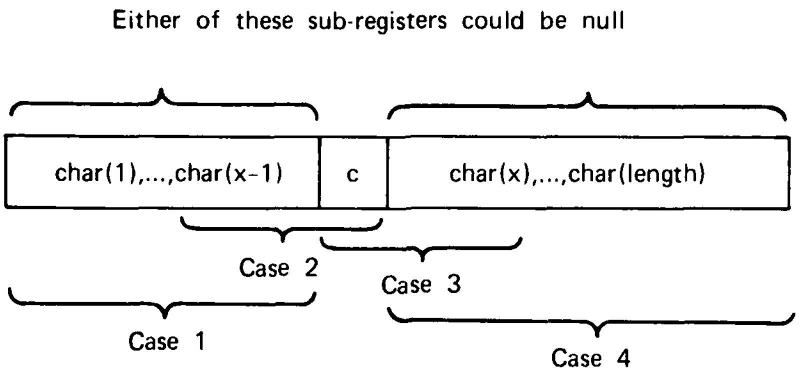
in addition to conjuncts (1)-(5). In this context we would try to prove

$$\text{LPRED}\ (1,\ \text{length},\ k,\ (\text{char}(k) \leq \text{char}(k+1))) \quad . \qquad (9)$$

However, from (5), (7), and (8) we can deduce

$$k < \text{length-}1,$$

in which case the range of k in (9) is subsumed by the range of k in (1). Thus (9) is true. Of the nine cases, five are trivially true because intervals in the LPRED are null. The other four cases are displayed in Figure 3.4. The ordered pair corresponding to each case indicates the output indices of the two conditional expressions. For example, the subcondition associated with the ordered pair consisting of the second and third cases (B, iii) looks like



Case 1: lpred $(1, (x-2), k, (\text{char}(k) \leqslant \text{char}(k+1)))$ (A,i)

Case 2: $\text{char}(x-1) \leqslant c$ (A,ii)

Case 3: $c \leqslant \text{char}(x)$ (B,iii)

Case 4: lpred $((x+1), \text{length}, k, (\text{char}(k-1) \leqslant \text{char}(k)))$ (C,iii)

*The ordered pairs indicate the relevant subcases of expression (6).

SA-2581-9

FIGURE 3.4 CASES IN THE PROOF OF A VERIFICATION CONDITION

$$\begin{aligned} \text{LPRED}(1, \text{length}, k, (k > x - 1 \\ & k = x - 1 + 1 \wedge \\ & k + 1 > x - 1 \wedge \\ & k + 1 \neq x - 1 + 1) \supset \\ & c \leq \text{char}(k)) \quad . \end{aligned}$$

This reduces to $k = x \supset c \leq \text{char}(k)$, simplifying further to the formula shown in Figure 3.4 (case 3). Thus the structure of the proof mimics the structure of the verification condition--something to be desired when automatic verification is attempted.

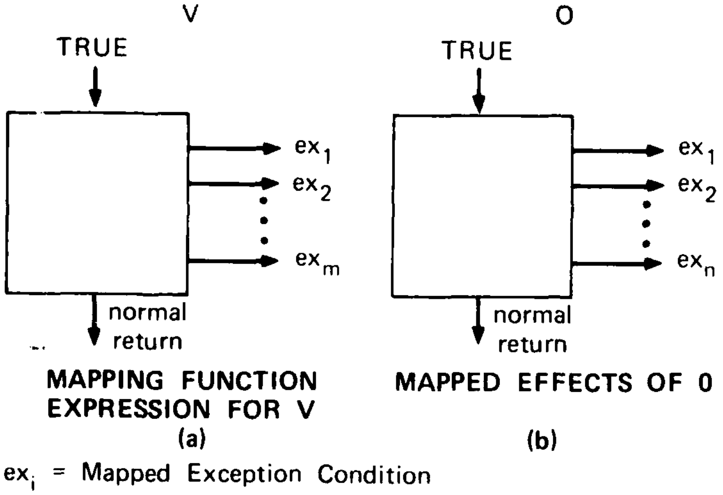
An automated verification system based on the hierarchical proof methodology need not have knowledge of the particular problem domains incorporated into its rules of inference. The abstract machine specification is all the domain information needed by the system. The deductive system of the current SRI verifier could perform this proof if it were given knowledge about the assertion language primitives, e.g., LPRED and conditional expressions. The program of Figure 3.3 can be enclosed within another loop to extend it to a full sorting program that resembles a bubble-sort algorithm. We have completed a proof of this extended program and compared it with a proof of a corresponding program, where the bubble sort is implemented in a nonstructured manner. The verification conditions for the latter are longer and more difficult to prove. We have also completed a proof of Floyd's TREESORT (Robinson[73b]) in which the comparison is even more convincing.

We are in the process of developing an assertion language that is well-suited to hierarchical proofs. Our language will be very simple, containing little more than integers, booleans, quantifiers, sets, and functions. The extension mechanism will be via abstract machines, which can be used to specify constructs not included in the assertion language. A final choice of an assertion language will also be motivated by the ease with which an automatic deductive system can process expressions written in the language.

Although automatic verification is a desirable goal, it may be impossible to achieve in practice. The deductive system might choose the wrong branch of a proof tree, leading to an infinite search, which is destined to fail. As a partial solution, it should be possible for the user of a verification system to observe and guide the proof process as much as he desires (or needs). This is the concept of a semi-automatic verification system, which we advocate for current work in program verification.

In stage 4, each implementing program has an entry point with no preconditions (i.e., an input assertion of TRUE) and several exits. The program can exist without a precondition, because it is assumed to have its own machinery for detecting exceptions and for reporting them back to the caller (via the multiple exits). If the function has n exception conditions, there are n + 1 exits, n for each of the exception conditions (with the mapped exception conditions serving as respective output assertions) and 1 for the normal exit. In the case of a V-function V (Figure 3.5a), the output assertion for the normal exit states that the program returns a value equal to the instantiated mapping function expression for V. In the case of an O-function O (Figure 3.5b), the output assertion for the normal exit consists of the "effects" section of O of the mapped specification for O.

Stage 5--Each of the primitive functions, both of M_0 and of the abstract programming language used in Stage 4, is implemented in terms of programs in an available well-defined implementation language, e.g., the



SA-2581-10

FIGURE 3.5 FLOWCHART DIAGRAMS FOR PROGRAMS IMPLEMENTING A V-FUNCTION AND AN O-FUNCTION SHOWING INPUT AND OUTPUT ASSERTIONS

instruction set of the hardware of a high-level programming language. Associated proofs concern the consistency of the implementation, e.g., of the compiler translation.

Note that stage 4 described the verification of a program containing exception conditions, by treating the conditions as multiple exits or return points. We must match the structure of a function's implementation by a corresponding structure in the calling program. We have come up with a control structure for a function call as follows:

```
⟨call⟩;
  begin call
    NORMAL: s0;
    EX1: s1;
         .
         .
         .
    EXn: sn
  end call;
```

The semantics of this control structure is to enter a CASE statement depending on the result of the call. Verification of calling programs would then be straightforward. However, this is very inconvenient for the programmer, who would have to account for a lot of unexpected exception conditions in the code. A suitable default structure, with routines generic to a particular kind of error, would reduce the programmer's effort, but keep the hierarchical formalism intact.

Another effort not directly mentioned as part of the methodology is the handling of concurrency in the proofs of the system. Several specific problems present themselves:

(1) proof of correct implementation of synchronization primitives of various levels of the system,

(2) proof of "indivisibility" of the functions of an abstract machine, and

(3) proof of properties of cooperating sequential programs.

One of the important steps in dealing with concurrency has been to be able to specify and prove properties of synchronization primitives and of processes containing them. Such work has been done in connection with our methodology by Robinson and Holt [73]. We have yet to precisely formulate the notion of "indivisiblity" for an abstract machine function, and the correctness of implementation for abstractly specified synchronization primitives. However, we can achieve proofs based on restrictions of these notions that will suffice for the operating system but we wish to establish the general criteria as part of the methodology.

In conclusion, we would like to emphasize that this hierarchical methodology for design, implementation, and proof is by no means a blind stab into software methodology. Besides the work of Parnas [72a,b,c,d, 74], the thesis of Price [73] developed the specifications as a design medium for part of an operating system. This virtual memory mechanism comprises the naming, addressing, protection, and sequential control facilities for an operating system. This was clearly a major achievement, and showed how the specifications could be used as a tool to present innovative ideas for system design. Furthermore, Price developed a technique for proving theorems, or global invariants, about a module specification. This resembles the generator induction principle independently developed by Wegbreit and Spitzen [75]. Parnas [72c] had performed some preliminary studies concerning the interconnection of several modules to form a software system--his KWIC index. Later work (Gerhardt and Parnas [73], Robinson [73a]) focused on the notion of a multi-level system using a hierarchy, but failed to capture any formal properties. Floyd's method [67] was first extended to the proof of programs containing O- and V-function calls by

Robinson [73b] in his proof of TREESORT. The idea of mapping functions (Robinson and Levitt [75]) was the last logical step to the formalization of the hierarchical methodology. Concepts similar to mapping functions have been independently developed by Hoare [72] and Spitzen [74], but their approaches concern individual data structures in a programming language. In their methods, precise specifications of each operation and access do not seem as important as widely-known mathematical concepts (the set of Hoare) or invariants (the stack and queue of Spitzen). The methodology appears to be gaining acceptance--both MITRE and Texas University will be using it in system design work, as well as other work within SRI. Other efforts, such as SOFTECH, are using a specification-related approach based partially on Parnas methodology.

In this chapter we have described a methodology for the design, implementation, and proof of large programming systems. The methodology localizes design issues to the relevant context, and it separates the issues of data representation and implementation. This localization and separation is attractive because it reduces the complexity to be dealt with at a given level. The methodology seems to lead to systems whose designs are understandable and whose properties are intuitively evident, even in the absence of proofs. We feel that the methodology is advantageous for proof because it reduces the proof of a large program to the proofs of numerous small programs, and because it simplifies the input and output assertions that are applied to each program. Due to the data abstraction provided by the hierarchy, the assertions tend to be expressed in terms of functions relevant to a particular level. Currently the main impediments to proving large programs involve the difficulty of framing assertions for the program and the difficulty of carrying out the deductions for large unstructured programs. We feel that this methodology holds attractive prospects for the proof of large software systems.

Table 3.1

REGISTER MODULE

PARAMETERS

i,j:	integer
c:	character

EXCEPTIONS

INDEX_RANGE(i): $i \leq 0 \vee i > \text{length}$
INSERT_RANGE(i): $i < 0 \vee i > \text{length}$
MAX_LENGTH: $\text{length} \geq 1000$

FUNCTIONS

```
V-FUNCTION  i = length

    PURPOSE:    returns length of register
    INITIALLY: 0
    EXCEPTIONS: NONE

V-FUNCTION  c = char(i)

    PURPOSE:     returns the character in place i of the register
    INITIALLY:   UNDEFINED
    EXCEPTIONS:  INDEX_RANGE(i)

O-FUNCTION  insert(i,c)

    PURPOSE:    inserts character c after position i in the register
    EXCEPTIONS: INSERT_RANGE(i);
                MAX_LENGTH
    EFFECTS:    ∀j (char(j) = IF j ≤ i THEN 'char'(j)
                              ELSE IF j = i + 1 THEN c
                              ELSE 'char'(j - 1));
                length = 'length' + 1

O-FUNCTION  delete(i)

    PURPOSE:    deletes the ith character of register
    EXCEPTIONS: INDEX_RANGE(i);
    EFFECTS:    ∀j (char(j) = IF j < i THEN 'char'(j)
                              ELSE 'char'(j + 1));
                length = 'length' - 1
```

Table 3.2

ARRAY MODULE

PARAMETERS

i,j,k: integer

FUNCTIONS

ARBOUNDS(i): $i \leq 0 \vee i > 1000$

EXCEPTIONS

```
V-FUNCTION  i = access(j)

     PURPOSE:   returns the value of the jth array position
     INITIALLY: 0
     EXCEPTIONS: ARBOUNDS(i)

O_FUNCTION   change(j,i)

     PURPOSE:    changes the jth element of the array to i
     EXCEPTIONS:  ARBOUNDS(i)
     EFFECTS:    ∀k(access(k) = IF k = j THEN i
                                ELSE 'access'(k))
```

Table 3.3

MAPPING FUNCTIONS BETWEEN REGISTER MODULE AND ARRAY MODULES

FUNCTION	REGISTER SPECIFICATIONS	MAPPED SPECIFICATIONS
length	φ: TRUE VALUE: length	φ: TRUE VALUE: ℓ
char(i)	φ: $1 \le i \le$ length VALUE: char(i)	φ: $1 \le i \le \ell$ VALUE: IF $1 \le i \le \ell$ THEN access(i) ELSE UNDEFINED

insert(i,c)	φ: (0 $\le$ i $\le$ length) $\wedge$
	(length < 1000)
	ψ: $\forall_j$(char(j) = IF j $\le$ i THEN 'char'(j)
	ELSE IF j = i + 1 THEN c
	ELSE 'char'(j - 1)) $\wedge$
	length = 'length' + 1
		φ: (0 $\le$ i $\le$ ℓ)
		($\ell \le$ 1000)
		ψ: $\forall_j$((IF 1 $\le$ j < ℓ THEN access (j)
		ELSE UNDEFINED) =
		(IF j $\le$ i THEN
		(IF 1 $\le$ j $\le$ 'ℓ'
		THEN 'access'(j)
		ELSE UNDEFINED)
		ELSE IF j = i + 1 THEN c
		ELSE(IF 1 $\le$ j - 1 $\le$ 'ℓ'
		THEN 'access'(j - 1)
		ELSE UNDEFINED))) $\wedge$
		ℓ = 'ℓ' + 1

delete(i)	φ: 0 < i $\le$ length
	ψ: $\forall_j$(char(j) = IF j < i THEN 'char'(j)
	ELSE char(j + 1))
		φ: 0 $\le$ i $\le$ ℓ
		ψ: $\forall_j$((IF 1 $\le$ j $\le$ ℓ THEN access(j)
		ELSE UNDEFINED) =
		(IF j < i THEN
		(IF 1 $\le$ ℓ THEN 'access'(j)
		ELSE UNDEFINED
		ELSE(IF 1 $\le$ j + 1 $\le$ ℓ
		THEN 'access' (j + 1)
		ELSE UNDEFINED))) $\wedge$
		ℓ = 'ℓ' - 1

Chapter 4

PROTECTION IN THE OPERATING SYSTEM

This chapter considers the protection of resources in the operating system. Section 4.1 is a brief tutorial on protection, with an emphasis on the notion of a type manager as the enforcer of protection rules within the system. Section 4.2 presents a description of a user-created type manager to implement bibliographies. Section 4.3 is a discussion of capabilities, presenting the advantages and disadvantages of their use as the basis of protection in an operating system.

4.1 Basics of Protection

In any operating system there is a distinction between objects, which are the resources provided by the system, and subjects, which are the entities that can request the system to manipulate objects. An object could be a segment, a file directory, or a program, for example. A subject is a computation (i.e., an instance of a program execution), presumably acting on the behalf of a user. A subject is also an object, since a computation can be created, started, stopped, or deleted (by some other computation). An object has a state, and a subject can obtain information related to or can effect a change in the state of an object, i.e., perform an access on the object. Different operations have correspondingly different access modes. Suppose that there is a function in the system,

$$\text{access: subject} \times \text{object} \times \text{mode} \rightarrow \{\text{TRUE, FALSE}\}$$

which states whether--for a given subject, object, and access mode--access is permitted. The mechanism that restricts the operations in the system to those permitted by the access function, and that updates the access function according to certain rules, is called the protection system.

Each object has a type. Objects of the same type have the same access modes. In general, the protection system may be centralized in the operating system, or distributed throughout the system structure according to types. In the latter case, which is adopted here, each type is supported by a set of programs called a type manager. In the operating system, type managers exist at every level of the system. In fact, the entire operating system can be considered as a hierarchy of type managers. Since a type is characterized by its operations and its state, we find it particularly convenient to specify the behavior of a type manager as a Parnas module.

In the set of types supported by the operating system, all objects are assumed to be primitive as far as specifications are concerned (although some objects, such as directories, are implemented in terms of lower-level objects). The operating system also provides the facilities for a user to create his own abstract types, choose the implementation of an object of each of these new types as an aggregate of objects of more primitive types, and code the procedures that implement the access modes for the new type. This is done by having a data base maintained by the extended-type manager that maps from names of objects to aggregates of names of more primitive objects. In the next section we describe the operation of a user-created type manager.

In summary, the type manager for a given type has exclusive control over the creation, use, and deletion of objects of that type. It uses capabilities as the primitive means of identifying objects of that type. The access modes for this type are interpreted by the type manager, which is the only type manager to whom they are meaningful. The use of levels of successive type managers adds to the usefulness of abstraction in a hierarchical design, providing just those operations required at the level of each type manager.

4.2 Bibliographies--An Example of a Type Manager

As an example of a user-created type manager, consider the bibliography example of Wulf et al. [74]. Here, the objects of concern are bibliographies, the subjects are programs desiring to perform operations on particular bibliographies. The possible operations are to:

- Create a bibliography,
- Add (delete) an item to (from) a bibliography,
- Merge two bibliographies,
- Print a bibliography, possibly without annotations that are private to the creator of the bibliography, and
- Delete a bibliography.

To perform any of these operations on a bibliography, a subject calls one of the set of programs that constitute the type manager for bibliographies. Before this program carries out the intended operation, it first ensures that it is being asked to manipulate a bibliography--and not an object of another type--and then ensures that the subject has suitable rights to the bibliography. The subject's rights are authorized if the subject is the creator of the bibliography, or if the subject retained the rights (through authorized channels) from another subject holding authorized rights. (As discussed below, a subject being authorized is equivalent to his holding a capability for the bibliography.)

The bibliography manager uses lower-level objects (supported by the system or possibly by other user-created type managers) to implement each of the bibliographies. The possessor of a right to a bibliography has neither rights to, nor knowledge of, the objects that implement the bibliography. Two reasons for hiding the representation of bibliographies are (1) that the type manager of bibliographies defines an abstraction that should not be circumvented by allowing users access to the representation; and (2) that a security violation might ensue, if user access to representations is allowed when multiple bibliographies share a common representation object.

It is relatively easy to use the generic facilities of a Parnas module to represent a type manager. The exception conditions for all functions visible at the interface of the module--O, V and OV--can check the rights of the caller. The V-function values can characterize the state of the objects maintained by the manager. For example, in the bibliography case, there would probably be V-functions defining for a caller (with adequate rights): (1) whether a bibliography exists (referenced by name), (2) the number of entries in an existing bibliography, (3) the contents of a particular bibliography entry--say, referenced by lexical position. The O-functions are the operations (given above) that a subject can call to modify a bibliography. The mapping functions between the bibliography level and the segment level characterize the representation of bibliography objects in terms of segment objects. (See Chapter 3.)

4.3 The Rationale for Capabilities

A capability is a token associated with a unique object in the system. A set of access modes is part of each capability. If capabilities are kept by a subject only in a particular structure (called a C-list), and can be used by subjects to gain access to the objects associated with them, possession of a capability by a subject implies the right to access the associated object in the modes specified by the capability. Since objects may be shared among many subjects, many capabilities (or copies of a capability) may exist for a particular object. Capabilities can be passed among subjects in various ways. Usually the possession of a capability implies that its owner has the right to pass it (or a copy of it) to other subjects. A simple capability system is a protection system in which the access function for a subject is TRUE for the objects and accesses indicated by the *capabilities stored in its C-list.*

Unfortunately, most interesting capability systems are not this simple. By allowing the storage of capabilities within objects, the description of the access function becomes much more difficult. However, this complexity is useful if a capability for an object is to survive the termination of its creating process. A much less restricted form of a capability system, defined below, is called a generalized capabili system.

Some form of capabilities exists in most operating system, particularly in virtual memory systems. For example, the descriptor segments of Multics (Organick [72]) and the virtual spaces, transfer vectors, and address space lists of Price [73] correspond to capability lists. However, there are restrictions on the manner in which these "capabilities" are manipulated. Furthermore, such capabilities apply to a very small set of types, and cannot be stored in objects that are not subjects. This prohibits type extension and the application of capabilities to more general protection problems such as data bases. Examples of more general systems are the Chicago machine (Fabry [67]), the Cal system (Lampson [69a], Plessey/250 (Cosserat [72]), HYDRA (Wulf et al. [74]), the CAP system (Needham [72]), and the system described here.

We shall refer here only to generalized capability systems. A generalized capability system is a system with a unified naming, access, and protection mechanism for objects. A generalized capability system has the following properties. (The first four are also found in simple capability systems.)

(G1) Every object in the system has at least one name (or capability). Subjects are entities that can store and use capabilities for objects.

(G2) Possession of the capability for an object is necessary (and in most cases sufficient) to guarantee access.

(G3) The mapping from capabilities to objects is a function.

(G4) Information defining a subject's permitted mode of access to an object is included in a capability.

(G5) Capabilities may be stored in objects that may be subjects or objects.

(G6) Capabilities may be passed to subjects and stored in objects, in some well-defined way. The passing of the capability for an object to a subject is thus the means for conferring access to that object upon the subject. The storage of the capability for an object in another object is a means for conferring access to the former object upon all subjects having access (in a particular set of modes) to the latter object.

(G7) There exists a function from objects to objects called the type function. The range of this function forms the set of types for the system.

(G8) The enforcement of protection (i.e., the interpretation of the access information contained in capabilities) is divided between the basic system, which guarantees some predefined rules for access to all objects whose type is among a fixed set of primitive types, and the subsystems, which can be programmed to guarantee some user-defined rules for access to all objects whose type is among a user-defined set of extended types.

Properties G1 through G8 are found in HYDRA and in the system described here, and to some extent in CAL. The important properties to prove about capability systems involve the rules for passing capabilities to subjects and objects (G6), and the rules for access to primitive types (G8).

The advantages of generalized capabilities can be summarized in three points:

- Generalized capabilities facilitate efficient dynamic creation of protection domains on a fine-grain time scale. This encourages the use of small domains to solve various special security problems.
- Generalized capabilities permit a powerful approach for attacking protection problems, namely the use of extended types. Interpretation of protection can be distributed in a uniformly controlled way.
- Generalized capabilities enhance the system by fostering a hierarchical system structure, by increasing the understandability of the system design, and by simplifying proof.

The operating system described here offers many features not found in other capability systems, making it a useful general-purpose facility.

In a generalized capability system, every resource in the system can be protected and accessed in the same manner. I/O devices, files, virtual memory, processes, and message buffers can all be considered as types. Accesses on objects of these types can be made by calling the appropriate _type manager_. The access can be implemented as a single machine instruction or as a procedure call to a program that implements the access, but in either case a capability is required.

Close to the notion of a type manager is the concept of _generalized domains_. At a given moment in time, a process executes in a domain (or environment) that describes the set of objects accessible to the process and the ways in which the process can access them. In many operating systems, a process operates in a single domain and can change only the components of that domain. This mode of operation is not very reliable, because different subunits of the program operate on only a small subset of the objects in the domain of the process. Certainly if a user program operated in the same domain as all system programs, the operating system would not be secure. Even having multiple user programs operating in the same domain violates the important precepts of modularity and need-to-know.

Multics has attacked the problem by allowing a process to operate in one of a set of nested domains called rings. A process can change its current ring by calling the operating system. Thus, the operating system can reside in the lowest (and most powerful) rings, whereas subsystems and user programs can operate in progressively higher rings so that they do not adversely affect the operation of the operating system. However, it would be advantageous to allow domains to be disjoint, in order to enable straightforward solutions to such problems as the Trojan horse problems and the mutual suspicion problem.

The call and return mechanism, as implemented in HYDRA and in the present system, can conveniently allow a process to switch control among possibly disjoint domains. A call switches control in a process from the domain of the calling program to the called domain, which is formed at call time. The called domain is the union of two domains--that of the pure procedure and that of the parameters. The return switches control in the process to the calling domain, and augments the calling domain with the domain of the returned values. Of course, domains for procedures, parameters, and return values are implemented via sets of capabilities. Thus, a pure procedure can contain the set of capabilities for the "own" objects of the procedure. These capabilities combined with the parameter capabilities and capabilities for objects created after the call (i.e., "local" objects), constitute the domain of a procedure activation. In the operating system, a process is considered to be a sequence of domains. In the implementation, each process corresponds to a stack of activations.

A domain is a useful construct for implementing a type manager in software. All extended types and some primitive types are implemented in this way. The presence of domains has important implications for the structuring and proof of the operating system.

The use of capabilities involves a general approach to the protection of objects in an operating system in which capabilities may be passed among subjects, but may also be revoked (see below). In authority-based systems, a single subject (or a fixed set of them) has control over an object's authority list. For example, suppose a subject "A" has access to an object. "A" cannot pass access to another subject "B" without the consent of the object's owner. Thus, delegation of tasks is very difficult in an authority-based system. In the case of capabilities, the possession of a capability normally implies the right to pass that capability to any subject "known" to the original subject. Even if the passage of the capability is restricted, it is an all-or-nothing proposition: either the capability can be passed to any subject, or to none. Thus, it is usually impossible for an object's "owner" to know exactly what subjects have access to the object. This fact hinders the centralized enumeration of the protection state, but makes possible the delegation philosophy for capabilities. This means that if a subject is trusted to access a given object, then the subject can also confer access to any subject that it trusts. This enables a subject to delegate to other subjects the tasks entrusted to it.

The passage of capabilities among subjects is in fact not unrestricted. It is possible to initialize a capability system in such a way that two subjects can never communicate, because they have access to disjoint sets

AD-A088 601   STANFORD RESEARCH INST MENLO PARK CALIF   F/G 15/3
A PROVABLY SECURE OPERATING SYSTEM.(U)
JUN 75 P G NEUMANN, L ROBINSON, K N LEVITT   DAAB03-73-C-1454

UNCLASSIFIED   NL

2 OF 4
AD A
088601

of objects and have no paths across which to transfer capabilities. Such paths are called capability channels. A message buffer and a directory are examples of capability channels, but even segments can act as capability channels if the hardware supports tagged capabilities that can be intermixed with data words in segments (as recommended here).

The operating system described here is structured hierarchically, in order to take advantage of the formal methodology for the design and proof of large system (see Chapter 3). Capabilities facilitate the use and enforcement of a hierarchical design. Programs at level n can reference only procedures at levels n-1 and below. Furthermore, a hierarchical structure enables a level to hide from higher levels those constructs that are available to it through lower levels. The "loss of transparency" can be easily implemented by not passing to the higher level at implementation time the capabilities to call the hidden functions.

Capabilities make assembly language programming possible and even enjoyable (Cosserat [74])! This saves us from the task of proving a compiler for a higher-level language as part of the operating system design. The reason for the ease of assembly language programming is that abstract operations are implemented through the call mechanism. Thus, the system structure provides data abstraction, leaving only control abstraction to be put into a very simple compiler/assembler to be used in the system.

Capabilities and hierarchical structure enable the distribution of operating system resources (and the protection mechanism) throughout the levels of the system. The hierarchical proof methodology of Robinson and Levitt [74] enables the structure of the proof to be distributed in a manner similar to the structure of the design. In addition, the number of parallel programs that have to be proved and the complexity of the proofs, are each reduced by the use of capabilities. This phenomenon, called data exclusion, means that a type manager does not have to maintain synchronization for disjoint capabilities. Since the representations for disjoint capabilities are presumably disjoint, then the representations are not in the same critical section and mutual exclusion is not necessary.

The division of a system into small domains increases understandability. Capabilities provide the information that links the domains (i.e., the capabilities "shared" by separate domains). Thus, capabilities are a tool for understanding the system structure in a manner analogous to a formal specification. In fact, capabilities are included in the formal specification language for the system because they are so primitive in the design.

The system design has permitted the solution of several "classical" problems in capability systems; namely handling revocation, preventing lost objects, facilitating small segments, achieving generality in protection, and achieving a capability-free user interface. These are discussed next.

A capability may not be withdrawn once it has been given away. However, it is possible to provide a mechanism called _selective revocation_ (Redell and Fabry [74]). If a subject anticipates the need to revoke a capability, a _revocable copy_ of the capability can be created. The subject can turn off the "revoke" bit on the revocable copy and pass that. In order to revoke the access conferred by the revokable copy, the subject calls "revoke(rk)" where rk is the revocable capability with the "revoke" bit on. The use of the revocable capability to access the object is disabled, leaving the function of the original capability intact (see Chapter 5).

The lost object problem arises when an object exists but all of its capabilities have been deleted. If lost objects can occur, system garbage collection must take place--an inefficient and potentially nonsecure operation. HYDRA solves the problem by having reference counts associated with each object. When a capability for an object is copied or erased, the reference count is incremented or decremented, respectively. An object is deleted when its reference count reaches zero. This is a low-level solution and is also inefficient. It involves the lookup of an object's representation (in a large paged hash table) whenever a capability for the object is referenced. Our operating system solves this problem by having a special type of directory entry called a distinguished entry. A directory entry contains a capability, and the directory system will not delete a distinguished entry unless the object corresponding to the entry's capability has been deleted. A higher level of the system forces a distinguished entry to be created for each object created above that level. Thus, at least one capability (the distinguished entry) exists for each user-level object in the system. Furthermore, the capability in a distinguished entry always has "delete" access, so that the object can be deleted as well as referenced.

In most capability systems, protection is quantized at a coarser grain than that of a single object (i.e., repositories for a small number of capabilities). The capability segments of CAL and the objects of HYDRA are examples of repositories for capabilities. The problem is that these repositories are objects themselves. Since the repositories are extremely small, there is a proliferation of small objects in the system. In a system designed to permit the use of extremely large objects, the proliferation of small ones rapidly creates havoc. This is especially true when each repository is a single segment. We attempt to ameliorate this problem by providing useful large data structures to hold many small sets of capabilities for short or long term. Examples of these are process stacks, bound linkage sections, the extended object mechanism, and directories (see Chapter 5).

Some protection problems are not easily solvable by capabilities alone. These are problems for which it is absolutely necessary to know the identity (or some attribute) of the caller, e.g., the military security problem. In most operating systems (including ours), the identity of the caller is needed for reasons other than the solution of high-level protection problems. Dispatching, scheduling, and process synchronization need to be able to identify the processes under their control. These functions need an authority-based mechanism to identify the process calling them.

One big question left largely unanswered by previous capability systems is, "Can capabilities present a reasonable user interface?" This question is unanswered because no generalized capability system has been running long enough to provide adequate statistics. Our system design provides some facilities, in addition to the basic required mechanisms, to facilitate user interaction. Extended objects, directories, and linkage sections are examples of such facilities. Extended objects do not occur in the CAL system, directories are not in the HYDRA kernel, and linkage sections are absent from both systems. Linkage sections and directories provide a complete facility for symbolic naming of system entities (see Chapter 5).

In conclusion, it seems that capabilities represent more of an implementation decision than a design decision in our system. After evaluating the goals of the project and the mechanisms needed to achieve these goals, we decided that capabilities were the best common denominator available. The most pleasant surprise, however, has been that capabilities can provide all of the features of a Multics-like user interface, and--at the same time--can meet the more general goals of the project.

Chapter 5

THE STRUCTURE OF THE SYSTEM

In this chapter we introduce the types of objects that are visible to a user of the system and the functions that manipulate objects of these types. We also discuss the design decisions made in ordering the managers for these types in a hierarchy.

A major challenge in design is to decide on an ordering of the types to satisfy optimally certain criteria, such as the efficient use of resources. Section 5.1 presents the visible types of the system. Section 5.2 defines the properties of the particular kind of hierarchy that has motivated the design, and discusses the implications of this definition on the different levels of an abstract system. The main aspect of the definition is that level A is above B if A is permitted to use the facilities provided by B. In Section 5.3 we present a partial ordering of the visible types, such that type A being placed above type B means that type A can effectively use the services of type B to implement itself. This ordering, in some sense, results from "obvious" design decisions. In Section 5.4 we discuss the final ordering of the types that results from resolving a few major conflicts in placement and from introducing a few hidden types. These hidden types, in almost all cases, are needed to provide more efficient operation. Specifications for the system are given in Appendix A.

5.1 The Visible System Objects

In this section we describe the desired properties of the objects visible in the operating system. The discussion will be confined to design decisions for single objects.

The objects are as follows:

(1) Capabilities, which provide the basis for protection as implemented by the type managers throughout the system. Capabilities are also a type.

(2) Segments, the major storage medium for the operating system. Segments can contain code or data, and thus can serve as procedures. Most file I/O is done using segments as files.

(3) User-defined and extended types, the mechanism whereby the user can create a type manager of his own.

(4) Directories, the permanent storage medium for capabilities. References to objects in directories are made by symbolic name.

(5) Linkage sections, a temporary storage medium for capabilities referenced by symbolic name. Each procedure activation has a separate linkage section.

(6) Processes and activations, the units of sequential control in the operating system.

Capabilities. Motivations and definitions for capabilities as the basis for protection are presented in Section 4.2. In our system, a _capability_ for an object is implemented as a two-tuple ⟨uid, access-vector⟩, where uid (the _unique identifier_) is a systemwide unique name for the object in question. For the purposes of this description, a given uid can correspond to only one object in the system. The _access vector_ is a boolean n-tuple, where each of the n-positions corresponds to operations that can be performed on the object. A TRUE in one position indicates that the corresponding operation can be performed, and FALSE that it cannot. The association of positions in the access vector with operations is done by the type manager for objects of the type in question. A FALSE in a given position in the access vector can never imply greater access than that of an access vector with a TRUE in the same position. Except for this restriction, the access vector bit patterns have no meaning apart from their interpretation by the type manager. If a subject possesses a capability with a uid corresponding to object x and a TRUE in the position corresponding to operation y, then we say that the subject has "y-access" for x, and equivalently, "y" abilities.

The possession of a capability by a subject is in most cases sufficient for accessing the pertinent object. Thus it is necessary for the system, in particular the capability manager, to protect the capabilities. A subject is not permitted to

- Create an arbitrary capability, since this might give him unwarranted access to some existing object. (A subject can call the capability manager to return a new capability which results in a new uid and an all-TRUE access vector. The capability manager can use the system clock or a counter to generate the uid's.)
- Modify the uid of some capability that he holds.
- Replace a FALSE in some position in the access vector by a TRUE. The replacement of TRUE by FALSE is an operation provided by the capability manager.

A subject in possession of a capability c can pass the capability via a channel for which he has a capability, e.g., via a shared segment or directory. Since the segment or directory might be readable by other subjects, this reoresents one approach for sharing in the system. A subject calls on the appropriate type manager to have an operation performed on an object. The calling mechanism involves the passing of a capability c as a parameter to the O-function (or the V-function if the state of the object is requested) that will carry out the operation. The subject's rights to the object, as embodied in c, are checked. The exception conditions in the function specification portray the type to which c must correspond, and the rights that are expected. For example, a call on the bibliography manager to append a bibliography represented by capability c would require that "append" rights be associated with c.

The specifications of the capability manager hide the implementation of the capability functions. However, we envision their being realized in hardware, since practically every instruction will require the processing of capabilities.

Revocable Capabilities--If an ordinary capability c has been delegated, the access thus permitted cannot be withdrawn without deleting the object to which it refers. A *revocable capability* c' for a capability c can be rendered ineffective by revocation without deleting the object, as follows. Given a capability c, a *revoking capability* cr can be created which refers to the same object, but which implies the ability to revoke any revocable capabilities derived from cr. A revocable capability c' may then be derived from cr by removing the ability to revoke. Although cr, and c' are all capabilities for the same object, revoking of cr ses invalidation of c' (and cr itself), but not c. (See Section 4 of Appendix A.)

Segments--One of the major design decisions was to provide virtual memory as the "memory" visible at the system interface. The primitive addressable unit in the virtual memory, visible to the users, is the *segment*. A *word* in a segment is identified by a name (i.e., uid for the segment) and an offset. The size of a segment can range from 0 to "maxsize".

The segment manager provides functions to create a segment of some initial size, change the length of a segment, delete a segment, write a word into a position of a segment, determine the size of a segment, and read the value of a position of a segment. When a user calls on the "create" function, he is returned a capability for that segment with the access vector containing TRUE in the positions corresponding to the operations "read", "write", and "delete". The creator of a segment can pass the capability to other users, possibly with reduced access rights. For example, a call on the segment manager to write on a segment requires that a "write" ability for a segment be passed as a parameter. If the "write" ability is not passed, or if other exception conditions are satisfied (e.g., the segment no longer exists or the offset is out of bounds), then the call is denied and an error routine is invoked.

The segment is the primary storage entity, representing virtual storage for procedure code, data and capabilities. In addition, the segment is the primitive object type from which more complex objects are synthesized. As demonstrated in Appendix B, each directory object is implemented as a segment.

This description hides the implementation of segments in terms of physical memory units. Another module (at a lower level than the segment manager) decomposes segments into pages and allocates pages in primary and secondary memory. The mapping from uids to page addresses is done in hardware, as is the rapid access to pages in main memory on the basis of segment uid and offset.

Extended-type objects--As we indicated previously, one important feature of the system is that users can establish their own type managers in order to provide a service for maintaining and protecting a particular class of objects. The operating system itself contains a module, called the extended-type manager, that supports the creation and maintenance of these abstract types and objects. The extended-type manager serves external users as well as system levels above segment manager. Each abstract into a position of a segment, determine the size of a segment, and read the value of a position of a segment. When a user calls on the "create" function, he is returned a capability for that segment with the access vector containing TRUE in the positions corresponding to the operations "read", "write", and "delete". The creator of a segment can pass the capability to other users, possibly with reduced access rights. For example, a call on the segment manager to write on a segment requires that a "write" ability for a segmenι be passed as a parameter. If the "write" ability is not passed, or if other exception conditions are satisfied (e.g., the segment no longer exists or the offset is out of bounds), then the call is denied and an error routine is invoked.

The segment is the primary storage entity, representing virtual storage for procedure code, data and capabilities. In addition, the segment is the primitive object type from which more complex objects are synthesized. As demonstrated in Appendix B, each directory object is implemented as a segment.

This description hides the implementation of segments in terms of physical memory units. Another module (at a lower level than the segment manager) decomposes segments into pages and allocates pages in primary and secondary memory. The mapping from uids to page addresses is done in hardware, as is the rapid access to pages in main memory on the basis of segment uid and offset.

Extended-type objects--As we indicated previously, one important feature of the system is that users can establish their own type managers in order to provide a service for maintaining and protecting a particular class of objects. The operating system itself contains a module, called the extended-type manager, that supports the creation and maintenance of these abstract types and objects. The extended-type manager serves external users as well as system levels above segment manager. Each abstract (or extended) object has a type and a representation in terms of more primitive objects, called representation objects. For a given abstract type, the correspondences between the objects and their respective representation objects constitute an object of type 'TYPE'. The type manager for objects of type 'TYPE' is the extended-type module. Thus, the extended-type module keeps a data base containing associations between abstract objects and capabilities (called implementation capabilities) for representation objects. For example, a bibliography object could be represented as one or two segments. In either case the association of a capability for each bibliography with the one or two corresponding capabilities is maintained by the extended-type manager.

Users can call the extended-type manager to create objects. To create a new type, a user calls "create_object," presenting the type-creation capability. The user receives a type manager's capability for the new type. When the new object is created, the user receives a capability for the new object. The type manager must then determine the representation for an abstract object by defining a tuple of implementation objects, which are represented by implementation capabilities. This can be done in two ways:

(1) by adding capabilities for already existing objects, or

(2) by telling the extended type manager to create new representation obj^cts.

These implementation capabilities are retrieved from the extended type manager by the appropriate type manager. Implementation capabilities may be freely added and deleted during the life of the object, thus supporting variable length objects.

Directories--The primary role of a directory is to act as a repository for capabilities. For each capability stored in his directory, a user can associate a name of his choosing. Above the directory level it is thus possible to refer to objects either by a symbolic name or by a capability. A directory consists of a set of entries, each of which is a two-tuple [name, capability].

In order to provide these services, the directory manager has functions for creating or deleting a directory object; for creating a directory entry, i.e., appending an entry to the set of existing directory entries; deleting an entry; retrieving all of the symbolic names in a directory; retrieving the capability associated with a given symbolic name; and moving an entry from one directory to another. Note that a directory can contain a capability for another directory.

The lost-object problem (Chapter 4) is solved by having a special type of directory entry called a distinguished entry, which cannot be deleted unless the object associated with its capability is first deleted. The problem can then be solved by having some higher level force all object creations to include the establishment of a distinguished entry as well (see Appendix A).

It is also desirable to have protection on a finer grain than directories, or else small directories will proliferate. Thus each directory entry has associated with it a set of uid's called locks. When a user wishes to get an entry out of the directory, he must present a key capability whose uid matches one of the entries for the directory, or a directory capability with "load" ability. Either one suffices.

Linkage sections--Linkage sections are short-term repositories for capabilities that allow per-process symbolic referencing of objects, and subsequently allow fast referencing via indirection through the linkage section. Linkage may thus be deferred until the last possible moment, i.e., when an object is first referenced by a procedure. There is one linkage section per calling procedure per process, and one entry for each called procedure.

Normal referencing of an object via the linkage section occurs as follows (Figure 5.1): in the source program, the reference is symbolic (e.g., "A"); the symbol table maps it to an offset in the linkage section ("offset"); that position in the linkage section contains a capability "c"; the capability refers to an object "obj". The hardware can use indirection through the linkage section to obtain the capability c just as if c were in a register.

When a compiler produces a symbol table, the list of symbols for external references is accumulated in a segment called a linkage template. Some of the symbolic references are directly resolved, so that a capability is placed in those positions. Those positions are called prelinked. Other references are left unresolved, with only a symbolic name in those positions.

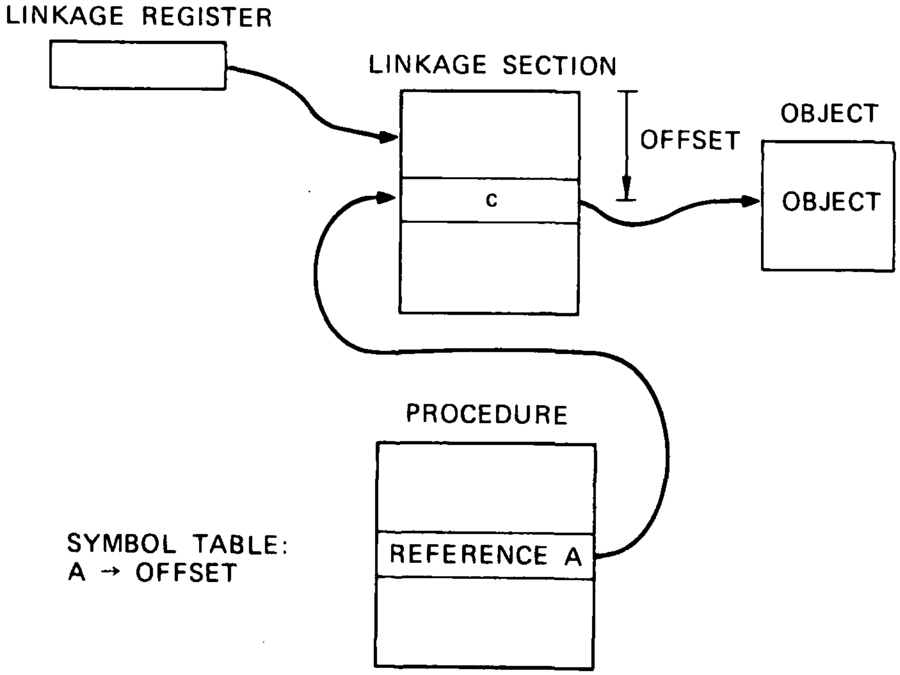


FIGURE 5.1 REFERENCING AN OBJECT VIA THE LINKAGE SECTION

Those positions are called *unlinked*. The first call from a given procedure results in the creation of a *linkage section* for that procedure, with one entry for each entry in the corresponding template. The linkage section contains capabilities from the linkage template in the prelinked positions, and some special symbol in the unlinked positions (perhaps a special tagged word). If the program attempts to use an unlinked reference, the hardware detects this (via the special tagged word) and creates a *linkage fault*. Control is passed upward to a routine called the *linkage fault handler*, which attempts to resolve the reference using the symbol in the linkage template. The reference can be resolved against a small data base for a user called the *name space*, or by searching through directories designated by the user. If the reference is resolved to a capability, the capability is inserted into the linkage section (i.e., the linkage section position becomes *linked dynamically*), and the reference is retried. Any prelinked or dynamically linked position in a linkage section is accounted for by the hardware as described in the previous paragraph. Linking can be done manually (under explicit program control) at any time after a linkage section is created. A reference may be unlinked (if it has not been prelinked) so that it can be bound to another object. A linkage section may be used (with or without unlinking) for repeated and recursive calls to the same procedure (in a single process). Thus dynamic linking may be required only once for an object used in many different calls to the same procedure.

The use of linkage sections and directories by the linkage fault handler makes available to the user a very large *virtual name space*, that can be easily used by programs with linkage sections. This facility is very similar to that provided by Multics. In a computational model related to block-structured programming languages, capabilities in linkage sections correspond to global or *own* variables; and objects created by the procedure and placed on the stack (see section on processes) correspond to local variables.

Processes and activations--A process is a unit of sequential control in an operating system (see Horning and Randell [73]). At any given point in time, a process has an address space or domain associated with it, corresponding to the resources currently under its control. Some conceptions of a process have a single domain associated with it, so that domains may be switched only by having inter-process communication. Multics provides a slight improvement, in which several nested domains called rings may be associated with a process. The operating system uses the innermost, or most powerful ring; subsystems use rings above that; and user programs execute in the outer rings. This domain structure is not adequate for solving the mutual suspicion problem (Schroeder [72]). We wanted a situation in which domains of execution could be disjoint, and in which parameters are passed in order to achieve communication. We chose a call and return mechanism to achieve this, because such a mechanism is relatively easy to implement in hardware, and because it is a naturally hierarchical one. Each call creates a new domain (an activation, or called domain), which is the union of a template domain (of the procedure called) and a parameter domain (the parameters passed to the procedure). Domains are easily implemented in our operating system by sets of capabilities. A process is considered to be a sequence of activations (implemented as a stack of activations).

Processes communicate with one another in a straightforward way using monitors (Hoare [74]). The protection associated with processes is interesting. Activations represent the smallest entity that can perform operations on other objects, so that activations were chosen as the subjects in the system. Processes can be created, deleted, stopped, and started. Activations can be examined (and perhaps changed) for debugging purposes. Access is permitted only if the process containing the activation is stopped, and the subject has a capability for both the process and the activation.

5.2 Our Concept of a Hierarchy

Our concern here is with defining the concept of a hierarchy of abstract machines, each machine of which is represented as a Parnas module. We present some general implications of a hierarchy on the design of an operating system. In a hierarchical ordering of Parnas modules, we say that module A is above module B if

(1) the V-functions of A (characterizing the data structures) are represented in terms of the V-functions of B, or if

(2) O- and V-functions of A can use O- and V-functions of B in their implementation.

(To be hierarchical, we do not allow B to be above A if A is above B.) Parnas [74] has contended that for all alleged hierarchies the parts and the relation among the parts should be defined precisely. In our hierarchy the parts can be (1) V-functions (of two or more levels), in which case the relation is "represents" (i.e., B represents A); or (2) O- and V-functions, in which case the relation is "implements" (i.e., B implements A). It is also possible to view the ordering as a calling hierarchy if we view the higher-level parts as "implementation programs for O- and V-functions" and the lower-level parts as O- and V-functions.

Another possible relation is "trust". Module A, in calling on a lower-level module B to manipulate an object, trusts B to perform the specified operations on the object and nothing more. A also trusts B to return control to A when B is finished. On the other hand, we do not wish to have B trust A. In order to enforce this criterion, it is necessary to ensure that the state of B can be modified only via the interface O- or V-functions provided by B. In Section 5.3 we discuss ways in which we ensure that this trust cannot be violated for particular modules in the system.

One additional relation is "knowledge of the semantics." That is, module B can have no information in its data base about the semantics of higher-level modules.

Many operating system experts have felt that it is not possible to design an operating system as a hierarchical ordering. Two primary reasons for this pessimism are (1) all systems require upward transfer of information, which "seems" to imply upward calls, and (2) hierarchical realizations have tended to be extremely inefficient.

Our solution with regard to complaint (1) is to allow upward transfer of control as a possible return from a call on an O- or V-function. Recall Figure 3.3 (for a general O- or V-function) which shows the entry and exit points. One exit point corresponds to a normal return, while the remaining exit points are in one-to-one correspondence with the exception conditions for the function. In the case of a return corresponding to an exception condition, we envision that the calling program provides a return address for each possible exception condition. The called module returns control to the appropriate address. The state of the called module is not altered by the call.

In order to illustrate the use of this upward transfer, consider a simple example that occurs in page fault processing. For a conventional system, assume a hardware level residing below a page-manager level. All instructions involve reference to a logical page and an offset within the page. The execution of machine instructions can be viewed as calling the hardware level. If a reference to a logical page cannot be handled by the hardware (because the page is not in main memory), the hardware must call the page fault handler--a call that violates the hierarchy. In our view of instruction processing for such a simple two-level hierarchy, all user calls to execute instructions correspond to calls in the page manager--not the hardware. The implementation of the page manager (hidden from the user) contains a table of the absolute location of all logical pages. Thus the page manager itself can call the hardware and either have the hardware return an exception if the logical page is not in main memory, or have the hardware successfully process the instruction. If control is returned to the page manager due to a page exception, the page manager can initiate a transfer of the referenced page to main memory, and then modify the page table. This modification of the page table involves calling on the hardware to modify the state of the hardware. The caller of the page manager is unaware of any of these hardware state changes.

The second complaint against hierarchies is that they lead to inefficient operation. This problem is partially mitigated by following the principle of grains of time. That is, the time required to process an instruction at level i should be significantly less than that for a level i+1 instruction. An order of magnitude difference has been suggested, but a 13 level system clearly precludes such a large difference. However, we have decomposed the system such that the most frequently used low-level instructions can be efficient.

One possible cause of inefficiency in a hierarchy results from the apparent need for multi-level interpretation of instructions, particularly those originating at high levels. This is true for system instructions that must filter down through many levels of implementation. In some cases the instructions appearing at multiple levels are almost the same.

In particular, our system provides "read" operations at three different levels. It would thus appear that there is much redundant mechanism among the three levels, and that all "read" operations from the user level would involve three nested calls. This phenomenon is known as duplication of mechanism, and has been the undoing of many hierarchical systems with regard to efficiency. We employ two solutions to this kind of problem:

(1) We allow duplication of mechanism with regard to specification but not with regard to implementation. That is, we have the implementation of all three levels of "read" instruction involve calls to the lowest, and most efficient level. Most of the higher-level calls can be handled at this low level. A "read" operation that needs higher-level software (e.g., a page fault handling) will trigger an exception, handled by the higher-level software (the page fault handling routine). The net effect would be as if the higher level had received the call.

(2) Where multiple nested calls are required for implementation where there is no duplication (such as the creation of an object), we provide an extremely fast hardware call mechanism (a single instruction) to make these nested calls efficient.

We thus avoid duplication of mechanism among levels of implementation, and minimize the time spent when higher-level functions must be implemented by multiple nested calls. Both mechanisms involve the incorporation of special hardware features, which are novel but well within the state of the art. This is one major advantage of designing the hardware as well as the operating system. In Section 5.3 we complete the hierarchy by introducing a few extra modules, mainly for efficient execution of basic segment operations and for the handling of processes.

5.3 Initial Ordering of Visible Types

In this section we present a partial ordering of the types introduced in Chapter 4 based on "obvious" design decisions. We are concerned with the following types: capabilities, segments, types (extended types), directories, linkage sections, linkage templates, processes, activations, monitors, and I/O devices. Besides these types there is a need for two visible functions to solve particular problems, namely the function that ensures that all objects are pointed to by at least one capability and the function that handles linkage faults and inserts capabilities within linkage sections. It is convenient to associate distinct system levels with those two functions. The ordering decisions of this section involve placing module A above module B if A can more effectively use the services of B for implementation than vice versa.

(1) Capabilities are at the lowest level. All type managers in the system enforce protection rules based upon the uids and the access rights associated with capabilities presented by a subject calling the type manager. Thus it is clear that capability creation, modification, and decoding should be at the lowest protection level.

(2) Segments are below any types that have large, variable storage requirements. A segment is the natural primitive object from which other objects can be synthesized. Revocation is handled at the same level as segments for symmetry of accessing.

(3) Extended-types should be used to implement some system types. Since the extended-type manager can maintain the representation capabilities for all non-segment objects, it seems natural to provide this service to all types, both system-defined and user-defined. The extended-type module should be above the segment manager, since it will use the segment mechanism to realize the data-base of representation capabilities.

(4) Directories are below the level that ensures each object has a capability. Since a directory is easily represented as an extended object, the directory module should be above the extended-type module. One way to solve the lost-object problem

is to place a capability for each object in a directory, such that its directory entry (a distinguished entry) is deleted only when the object has been deleted. Thus, the lost-object module uses the directory module, and must be above it in this solution. Also, object creation is used by the lost-object module so that all object creation must take place below it, no higher than the extended object module.

(5) Linkage sections and directories should be below the linkage-fault handler. The linkage-fault handler will search appropriate directories in order to locate an entry that contains a presented symbolic name. The capability associated with this entry is placed in the linkage section by the linkage-fault handler.

(6) Processes and activations are extended objects. It is convenient to realize both a process and the activations within a process as extended objects, thus the process module should be above the extended-object module.

(7) I/O device signaling should be above monitors. It is convenient to use a special type of monitor to control the communication and synchronization associated with I/O devices. Thus an I/O operation is manifested as either an attempt to enter a monitor, or to do a "visit" or "signal" on a condition variable of a monitor.

5.4 Resolution of Conflicts in Module Placement

In Section 5.3 we produced a partial ordering of the principal system types. In this section we identify our design decisions that led to the decomposition shown in Table 1.1.

5.4.1 Segment addressing within the virtual memory. At the interface of the segment module, functions are provided to perform basic manipulations of segments: read, write, change segment size, etc. The user is able to view a segment as a block of virtual memory without regard for the underlying mechanisms associated with addressing the segment, e.g., its actual location on physical storage. One important aspect of the implementation of segments is the allocation of pages of main store and secondary store that hold the segments. We have decided that a segment can reside in one or more pages, but a page will never hold the contents of more than one segment. For convenience in addressing, the allocation of a segment to pages is as follows: if the page size is p, the first p words of the segment are allocated to page 1 for that segment, the next p words to page 2, etc. If the last page is not filled to capacity, a bounds-check mechanism prevents any accesses beyond the limits of the segment on the last page. (If we discover at some later time that many small segments will occur, then we might change the above decision.)

With regard to the handling of paged segments, we have decided on three levels of segments:

- (user) segments (level 4), which provide a variable size virtual memory. The segments visible above level 4 are these active segments.

- fixed-VM segments (level 3), which provide a small virtual memory that remains fixed with respect to the number of segments and size of each segment. As we discuss below, fixed-VM segments are used only to implement the level 4 functions.

- main memory pages (level 0) which provide a fixed physical memory.

The objects of level 4 are all of the segments that are currently active, i.e., that have been created but not deleted. Level 4 maintains a data base, not visiblc at the module interface, that associates with each segment uid a location area in secondary storage where the segment is found, and a main memory page location for any pages of the segment that are in main memory.

The objects of level 3 (fixed-VM segments) are used only for holding the data bases and procedure of level 4. For all such segments, there exists a segment in main memory that gives the correspondence between the fixed-VM segment uid and its location in secondary memory. This property guarantees that no more than one page fault will occur for any reference to a fixed-VM segment, or equivalently for any instruction in the implementation of level 4. (See also Saxena and Bredt [75].)

The objects of level 0 are pages of segments (i.e., subsegments) that are in primary memory. In order to maintain these objects, level 0 has a data base that gives the main memory page address for each page in primary memory. Since this data base is accessed every time a segment is accessed, it is desirable to keep a portion of it in very fast memory, e.g., associative memory. The data base is modified by levels 3 or 4 in transferring a page of virtual memory from secondary to main memory. It is clear that level 3 data base and the programs of level 3 will reside in main memory segments. Other segments, i.e., fixed-VM segments or segments belonging to modules above level 4, will become main memory segments as they become referenced.

The three levels 0, 3, and 4 constitute a three-level memory hierarchy. A two-level memory hierarchy would suffice for situations where mapping tables that hold the location of each unit of virtual memory could be maintained in main memory. However, in our system we envision that the number of segments will exceed the main memory space, hence requiring that mapping tables themselves be paged.

Each of the levels provides operations at its interface to reference and modify the contents of segments of that level. For example, each level provides "read" and "write" functions. The system would be intolerably inefficient if a call on the level 4 read was first interpreted (e.g., by software) as a level 3 "read" which in turn was finally interpreted as a level 0 "read." We provide an efficient implementation of the basic segment operations that avails the multi-level interpretation. Chapter 7 discusses this and other implementation decisions.

5.4.2 Fixed vs. variable number of processes. One of our basic design decisions is to accommodate a potentially large number of processes. There is a large number of user processes, plus a small fixed number of processes to handle I/O devices and the system clock.

A given process can also create other processes as needed. We say a process is dispatched if it is currently executing instructions on some processor--a cpu or possibly an I/O processor. A process is not in the "dispatched" state if it is blocked on some condition, e.g., a page fault.

If a dispatched process suffers an interrupt, there should be one or more processes that can be readily dispatched. It is clear that a process is a candidate for being rapidly dispatched only if some information associated with the process is in main memory. This main memory information must contain

- the status of the process including the location of next instruction, and
- a location of this status information. (This knowledge of the process is maintained by the process manager.)

It is not possible to allocate sufficient main memory to handle a potentially large number of processes. Our solution is to have two separate levels concerned with process management. Level 2, called the scheduled process manager, manages a fixed number of processes and controls the dispatching of these processes. All of the segments that implement level 2, in addition to some information about the fixed number of processes that level 2 manages, are in main memory.

There is a need for another module, which we call the scheduler, to manage all of the user-created processes. (The need for two levels to accommodate a variable number of processes is also discussed in Bredt and Saxena [74].)

The scheduler is called by users for all operations or processes, e.g., "create", "block", "kill"; the scheduler also maintains the activations associated with the processes it manages. To make a process dispatchable, or to block or stop a dispatchable process, the scheduler will call level 2. It is obvious that this module must be able to use the services of the variable size virtual memory, which means it must be above level 4. There are conflicting arguments for placing the scheduler immediately above level 4 or at a high level of the operating system. The next paragraph discusses these conflicts and the reasons why we favor the choice of level 10.

We must design the two levels of processes so that the higher-level process manager (level 10) never gets control of a process that is active at level 2. In addition, when a process executing at level 2 is stopped by level 10, the state of the stopped process must not contain information about activations between level 2 and level 10. This would violate the hierarchy, allowing level 10 to inspect and change the states of lower levels without calling them.

Chapter 6

USE OF THE SYSTEM

The purpose of this chapter is to provide an overview of how the system appears to its users. Two views are given--the view from the user-visible operating system functions (level 10), and the view from command level (here considered as level 12). The establishment of initial user authorization is discussed as part of the "login" command.

6.1 User-Visible Operating System Functions

The set of operating system functions visible at the operating system interface (level 10) is given in Table 6.1, along with the level at which the function is defined. Descriptions of these functions are found in Appendix A, along with specifications of these and other functions at each level.

6.2 Commands

A command is simply a procedure or set of procedures conforming to some collection of command standards, e.g., for arguments, macros, and symbolic error returns. Some commands directly reflect lower-level functions (e.g., "restrict_ac" or "create_process"), or provide other useful system functions such as "login", "logout", or "print". Additional commands provide user-oriented functions such as compiling, editing or debugging. Command level hides most of the details concerning the implementation of the operating system, e.g., directory formats, linkage sections, object creation, and perhaps even capabilities.

A summary of various illustrative commands is given in Table 6.2. For purposes of the examples, the argument "name" is a symbolic entry name or a path of entry names relative to some fixed directory, except in the case of "ioname", which is a symbolic stream name representing a device or pseudodevice. (This command set bears some resemblance to Multics, for descriptive simplicity.)

As an example, the input-output commands can be thought of as those of the Multics system (Feiertag and Organick [71]), with dynamic attachment of ionames to other ionames. Some ionames are dedicated to particular devices, and some represent type managers for pseudodevices such as a deferred printer, or a logical formatter, or a multidirectional broadcaster (which in turn issues replicated "write" calls for different devices).

Another example is the "delete_object" command, including deletion of objects of type "segment" and "directory". Various forms of deletion are desirable. For example, a user may wish an object to disappear instantaneously (with potentially annoying effects on current users), or when no longer in use. He may wish to rename an entry for that object in a directory, so that new users will not be able to link dynamically to the object by its expected symbolic name. By renaming an entry with name n (representing capability c, in directory d) to n.old, for example, and deleting any object previously named n.old, a "deferred delete" of depth one is obtained. Deferred deletes of specific time delays (e.g., until 4 AM,or after 24 hours) can also be obtained in such a way that the object referred to by n.old (with capability c) is no longer available to any user except those already dynamically linked to it. Thus the "delete_object" command may have a variety of options associated with it.

6.3 Command Interpretation

Command interpretation involves the expansion of command macros, and the invocation of any systemwide or user-imposed conventions. Command interpretation takes place at level 11. Examples of what may occur at this level include:

- interpreting delimiters for deconcatenating command streams
- argument conventions regarding interpretation of directory names, i.e., relative to the visible root, or relative to the working directory
- expansion of macro commands, and substitution of interpreted values for expandable argument macros
- argument conventions regarding access control list specifications, e.g., with known sets of users and multipart identifiers
- interpretation of return arguments, especially with regard to error conditions.

(The Multics command interpreter is a good model for what might happen here.)

A user may provide his own command interpreter if he is unsatisfied with the system standard one. However, a user should beware of accepting someone else's private command interpreter, since this provides a potential Trojan horse condition.

6.4 Initial Authorization

As far as security is concerned, the starting point for discussion is the "login" command. Logging into the system requires some form of user identification--a password, a palmprint, the maidenname of some maternal grandmother, or perhaps the physical identification of being recognized by a security officer. As far as the operating system design is concerned, such identification is outside the design, although the integrity of the identification is of course itself subject to protection as if it were a part of the system. Thus, for the present discussion, successful identification and authorization of login is assumed. At this point, the crucial issue is how to specify the initial authorization to be given to the user whose identity has just been verified.

Initial authorization thus reduces to the question of what capabilities are made available to the user. Many of the issues that arise here are policy issues, which will differ from one installation to another. However, the basic mechanism remains the same. At least one process must be created (or allocated) in which the user may execute. Each such process must be given an initial set of capabilities suitable for his authorization. The minimal set involves capabilities to access the process stack and a working directory appropriate to the login request. Capabilities to access some subject of system library directories may also be appropriate, depending on how restricted the user's access is to be. In principle, any machine instruction may itself require a capability, although this is an extreme policy, and might be invoked only for a few instructions. Initialization may also set up user profiles containing private standards, abbreviations, and protocols.

Once a user process has been initialized and returns to command level, it is then ready for executing user commands. From then on, the notion of authorized access (either acquisition of information, or alteration of information) is explicitly related to the capabilities that may be obtained by using the system. Thus capabilities provide the basis for the statement and proof of properties of security (see Chapter 8).

6.5 Special User Subsystems

As noted above, the process created for a user at login may be given an initial authorization that is arbitrarily restrictive. One example of interest here is that of a secure document manager, supporting a multi-level classification system (top secret, secret, etc.) with categories and "need-to-know." In order to assure sufficient compartmentalization of different levels, it may be desirable to make a new login necessary in order to change the working classification level (and the corresponding working directory). This example is considered in greater detail in Appendix C.

Table 6.1

USER-VISIBLE OPERATING SYSTEM FUNCTIONS

Level	V-Functions	O-, OV-Functions
10	j = read_env (i,c)	c = create_process (st) start (c) stop (c) delete_process (c) call (f,n) return (n) push (f,n) pop (f,n) write_out_of_env (f,i) write_into_env (f,i) set_upper_bound (n) write_env (f,i,c) decrease_env_length (c,n) append_env (f,c) c = create_monitor (cm,n) delete_monitor (c) enter_monitor (c) exit_monitor (c) wait (c,cv) signal (c,cv)

8	b = ℓtemplate_exists (ℓt)
	j = get_size (ℓt)
	b = ℓname_defined (ℓt,i)
	n = get_ℓname (ℓt,i)
	b = prelink_exists (ℓt,i)
	c = get_prelink (ℓt,i)
		ℓt = create_ℓtemplate (tℓ,j)
		delete_ℓtemplate (ℓt)
		define_ℓname (ℓt,i,n)
		prelink (ℓt,i,c)

7	c = get_cap (d,n)	c = create_object (d,n,t)
		s = create_segment (d,n)
		dl = create_directory (d,n)
		delete_object (d,n)
		delete_segment (d,n)
		delete_directory (d,n)
		ck = create_revocable_cap(dk,nk,c)
		revoke(ck)

Table 6.1 (Concluded)

Level	V-Functions	O-,OV-Functions

6	b = valid_dir (d)
	{w} = get_locks (d,n)
	b = entry_exists (d,n)
	{n} = dir (d)
	i = dir_size (d)
	i = lock_set_size (d,n)
		create_entry (d,n,c)
		remove_entry (d,n)
		move_entry (d,n,d1,n1)
		add_locks (d,n,k)
		remove_locks (d,n,w)

5	i = object_type
	b = exists (c)
	b = initialized (c)
	[c] = impl_cap (c1)
		c = create_impl_object (c1,c2,c3)
		insert_impl_cap (c1,c2,c3)
		delete_impl_cap (c1,c2,c3)

4	i = read (a)
	i = seg_size (c)
	b = seg_exists (c)
	u = chase_ind (c)
	u = real_ind (c)
		write (c,j,i)
		change_seg_size (c,i)

0		c = create_cap
		c1 = restrict_ac (c,i)

Table 6.2a

COMMANDS DERIVED DIRECTLY FROM LOWER-LEVEL FUNCTIONS

Level Used	command_name arguments
10	quit /GO TO quit_handler/
	create_process name /for additional processes/
	delete_process name
8	link procedure_name called_name called_tree_name
7	create_directory dir_name
	delete_directory dir_name
	copy_directory dir_name
	copy_segment name
	create_segment name
	delete_segment name
	create_object name type
	delete_object name
	create_revocable_name c_name k_name
	revoke k_name
6	list directory dir_name options /locks are an option/
	copy_entry name new_name
	rename_entry name new_name
	move_entry name new_name
	remove_entry name
	add_locks name key
	remove_locks name key

Table 6.2b

OTHER ILLUSTRATIVE COMMANDS

```
                    command_name arguments

login user_name  /options/

logout  /options/

"editor_name" source  /edit_requests/  /options/

"compiler_name" source_name

"name"  /arguments/
         [/note:  this executes a program named "name"/]

"debugger_name" source_name  /options/

bind object_name object_name1 object_name2

change_working_directory new_dir_name

"io_function_name" ioname  /args/
                           /cf. Multics command "iocall"/

         [e.g.,
          print ioname  /options/;
          attach ioname ioname1  /options/;
          read ioname  /options/;
          write ioname  /options/;]

send_mail

open_mail

execute_in_memoryless_environment
```

Chapter 7

IMPLEMENTATION CONSIDERATIONS

This section presents an approach toward producing an implementation of the operating system. Since most of the effort on implementation is beyond the scope of the work described here, the discussion of this chapter should be treated as preliminary. Its main purpose is to illustrate the techniques for expressing representation and implementation decisions within the hierarchical methodology, and to give a few of the implementation decisions that have been confronted.

Section 7.1 discusses briefly the meaning of representations and implementations within the methodology. Section 7.2 discusses implementation decisions. These relate primarily to ensuring efficient operation of the system. Section 7.3 presents some preliminary thoughts on possible hardware that matches some of the design decisions. Most of the suggested features support the special protection mechanisms to avoid severe software interpretation penalties.

Appendix B gives examples for the system specified in Appendix A. In particular, Section 1 of Appendix B illustrates the methodology by presenting a representation of directories in terms of segments and objects of type "type", i.e., the representation of level 6 in terms of levels 4 and 5. Section 2 of Appendix B gives the representation of level 5 in terms of level 4.

7.1 Methodology for Representation and Implementation

As we indicate in Chapter 3, representation and implementation are confined to Stage 3 and to Stages 4 and 5, respectively. In Stage 3 we represent the data structures of level i in terms of level i-1 (and possibly lower levels) by writing mapping function expressions for each V-function of level i. Each such mapping expression is an assertion relating a particular V-function of level i to V-functions of the appropriate lower levels.

These mapping function expressions bind some of the representation decisions, but not all of them. For example, in Section 1 of Appendix B we give mapping function expressions for the V-functions of the directory module that represent the set of entries in a directory as a linking of words in a segment. However, the mapping function expressions do not bind the order of the elements of the list. That is, any ordering within the list will satisfy the mapping function expressions.

In Stage 4, abstract programs are written to implement the O- and V-functions of each level i ($i > 0$) in terms of level i-1 (and again possibly lower levels). These programs complete the binding process begun in Stage 3, and remove any of the nondeterminacies illustrated in Figure 3.2(c).

We envision that each level of the system will be shared by many users. That is, each of the O- and V-functions can be subjected to a call by one user before the processing of the call on behalf of another user is completed. Each module is required to enforce mutual exclusion rules that are appropriate to its resources. We envision that a module provides a program implementing an O-function with exclusive use to the V-function values that are subject to change by that O-function. Calls on O- and V-functions are allowed to proceed provided they do not require access to any of these V-function values. The calls that are not allowed to proceed can be deferred, but in a manner hidden from the caller. The monitor facilities of level 2 can be used by the operating system modules to implement the synchronization of function calls.

If users wish to create their own subsystems, and represent them as a sub-hierarchy of modules, they can use the monitor facilities of level 10 to achieve synchronization and exclusion.

Another issue with regard to implementation of the modules is the handling of error calls. At present we have not selected a mechanism for this, but (as noted in Figure 3.5) we envision that a call on a module will include in its parameter list the return addresses for each possible error condition anticipated, in addition to a return address for the normal return. Level 2 provides the functions for calls between system levels. The level 10 "call" serves the same role for calls initiated above that level (e.g., user calls). As we discuss in Section 7.3, we envision significant hardware support for all calls.

7.2 Sample Implementation Decisions

In this subsection we outline some of the important implementation decisions that we have confronted, in the course of developing the design. Although we have suggested that in the methodology the design logically precedes the implementation, it is sometimes essential to keep in mind when specifying the module how a module is to be used. We present here some of the decisions on how a particular module makes use of another module, although we urge the reader to keep in mind that these decisions are subject to change.

Handling of interrupts in a chain of indirect addresses--The purpose of level 0 is to execute instructions that require a small fixed number of memory accesses, for example to write the contents of one memory word into another word. These instructions are indivisible with regard to any interrupts. We envision a basic instruction set in which an arbitrary number of indirections might occur for a given instruction set. We do not wish to defer all interrupts until the completion of an instruction that might incur a long sequence of indirections. Hence, level 1 provides an abstraction in which arbitrary indirections can occur. The implementation of level 1 involves a sequence of calls on level 0, with interrupts possibly occurring at the completion of each call.

Efficient handling of basic segment operations--As indicated in Chapter 5, all user segments are accessed via level 4, although levels 3, 1 and 0 all support segment operations--the latter two levels supporting segments in main memory. Since practically every user instruction will be manifested as a segment "read" or "write", it is necessary to implement the level 4 "read" and "write" as efficiently as possible. Although the basis of the implementation scheme for functions is the hierarchy, it appears that a multi-level interpretation of these operations occurs. However, it is possible to implement the functions, so that in most cases they are interpreted directly and rapidly in hardware.

Consider a user-initiated call on the level 4 "read" function. A portion of level 4 in hardware maps this to a level 1 read. Level 1 determines if the referenced page is in main memory by consulting the mapping table, a portion of which will be in high-speed memory. It is also determined if "read" abilities are presented. If these checks are successful, as they should be most of the time, level 0 read is called and in most cases a value is returned within a single machine instruction.

On the other hand, if an entry is not found in the mapping table for the referenced segment, there are three possible causes:

- There is no segment corresponding to the uid presented.
- The requestor's access rights have been revoked.
- The referenced page is on secondary store, signifying a page fault.

With regard to the last alternative, an error trap provides return of control to level 4, which implies that now a level 4 instruction is to be executed. This will precipitate a call again to level 1, but this time on behalf of level 4. If the call on level 1 and the resultant call on level 0 are successful, then the processing of the page fault is initiated. Ultimately the referenced page will be brought into main memory, the mapping tables updated and the original read on level 4 completed.

However, the trap return instruction, or any succeeding instruction can precipitate a page fault. In this case, the trap return goes to level 3, which is guaranteed to have the location of the referenced page in the level 3 data base. Thus level 3 initiates a transfer of the page needed by level 4 to main store, and level 4 can continue its processing of the original page fault.

Thus we see that most segment operations will be efficiently handled in one machine instruction. Those that require access to secondary storage will incur upward transfers that do not propagate above level 4. The user is thus unaware of the implementation of his instruction. The operation is handled by interpretation through level 4 whenever a call on level 4 is not handled initially by levels 1 and 0. Level 3 is invoked whenever level 4 processing incurs a page fault.

7.3 Preliminary Hardware Considerations

Although the specification of the hardware is a major task of the next phase of our work, some of the hardware requirements have become evident as a result of some of our design decisions. Among these are the following:

Capability creation and manipulation--The entire security of the system is predicated on the nonviolability of the capability mechanism. Hence special hardware is needed to prevent the conversion of a data word to a capability, the modification of the uid of a capability, the conversion of 0's in the access vector to 1's. Since capabilities will be interpreted on every instruction execution, it is clear that their interpretation must be rapid. A convenient approach is to have all capabilities be specially tagged and provide hardware to perform special logical operations on words that have the tag present. The system clock can be used to generate new values of uid.

Mapping tables--As indicated in Section 7.2, level 0 contains tables that give the correspondence between the segment uid and offset and the page location, for all pages in main memory. It is clear that this table will be referenced by every instruction. A portion of this table should be in very fast memory, say associative memory to expedite the access. The remainder can be in main memory.

Call and return instructions--One defect of current capability-based systems is the excessive overhead associated with the call instruction. In our system a call on a module is a frequent occurrence, particularly for low-level modules. There is a significant overhead expense associated with such calls, involving the checking of exception conditions, passing of parameters, and the creation of new activation. We envision instructions in the hardware to support all of the operations associated with "call" and return".

Tagging--There are several benefits to designing a processor that recognizes tagged capabilities. Most of these are related to increased efficiency. Efficiency is not a trivial problem, because most operating systems in which capabilities are maintained by software are almost prohibitively slow. An implication of tagged capabilities is that capabilities and data may be freely intermixed. This eliminates the overhead involved in maintaining capabilities and data separately. (In HYDRA, every object must have two parts: a capability part and a data part. In the CAL system, for example, capabilities may be placed only in a special capability segment.)

Another advantage to hardware recognition of capabilities is that capabilities can be used in address translation without an interface to any other representation. Address translation includes linkage in our system. Most machines have hardware-assisted relocation. There must be a special interface to this mechanism if capabilities are implemented in software. The interfaces between the system and both I/O devices and process representations are also improved if the hardware can directly process capabilities for both processes and I/O devices. Not only can the system be faster, but it can also be smaller, because there is less mechanism to be programmed.

Another important reason for hardware recognition of capabilities can be found in the call and return mechanism. In HYDRA, on the average, 300 instructions are executed for each call. The system described here provides a single-instruction "call". At this point in the design this call omits the relatively small effort needed to save registers and push parameters (operations done by the HYDRA call), purposely leaving the "calling conventions" up to the language designer for now. (Saving and pushing may at a later date be added to the "call", if this is desirable.) This makes it feasible to have many small domains--a situation that is tolerated but not encouraged in other capability systems.

Certain types may also be recognized in hardware. Candidates include segments, stacks, and revocable capabilities.

For an efficient hardware realization of the system design described here, it is expected that all of the functions of levels 0 and 1 would be hardware instructions, along with the traditional processing instructions (arithmetic, logical, etc.). Also all error conditions at levels 0 and 1 are implemented as hardware traps. In addition, other functions of higher levels may advantageously be implemented directly as hardware instructions, or via special hardware instructions not required by lower levels. Examples include:

level 2: clock operations, an indivisible lock instruction (similar to the "real_alter_rewrite" Multics STAC instruction on the 6180), call, return, push, pop

level 3: multilevel storage operations, input-output (relocatable and subject to the protection of capabilities);

level 4: read, write;

level 10: call, return, push, pop, write_out_of_env, write_into_env (the latter two are forms of "move").

Furthermore, instructions supporting monitors for levels 2 and 10 are also feasible.

Less efficient realizations may be achieved by implementing the functions of levels 0 and 1 interpretively or via microprograms, using existing hardware.

It should be remembered that the functions of the user-visible interface (Table 6.1) are those that are of primary interest in considering implementations. The nonvisible functions included here could be changed without affecting the user-visible interface, and thus represent a possible internal system design. In that certain higher-level functions may be implemented directly in hardware, as noted above, some of the nonvisible functions specified in Appendix A may actually be changed by or disappear into an actual hardware implementation. Thus different systems belonging to the family specified in Appendix A may have widely differing implementations. In addition, each system may have its own command structure, with correspondingly different user interfaces and efficiency of implementation.

Chapter 8

SECURITY ASSERTIONS

8.1 Introduction

This chapter presents a preliminary discussion of what it means for an operating system to be secure, and how security properties of the system can be proven. Section 8.2 discusses the meaning of security in intuitive terms. In particular, it summarizes what we believe is the strongest possible concept of security in terms of four principles that relate to the nonoccurrence of certain undesirable events, and a set of variant assertions that relate to changes in the system's protection state as a result of operations. If it can be shown that the specification of each function in the operating system satisfies the principles and the variant assertions, then the design is said to be secure. Intuitive justification is given for the completeness of these principles in ruling out various types of penetration. Section 8.3 presents the first two principles in formal terms, relating to the unauthorized alteration and the unauthorized detection of information. These two principles, slightly generalized, are satisfied completely by each of the functions of the operating system, with the exception of a few cases where the specifications as given in Appendix A need to be modified. The generalizations and the exceptions are discussed in Section 8.4. Principles P3 and P4 have not yet been formalized. However, Section 8.5 provides a discussion of tentative approaches to formalization of the confinement principle (P4). The confinement principle is not satisfied completely by the system (nor is it theoretically attainable in a complete sense), but there is justification for asserting that the communication channels are of very low capacity.

As is shown below, these four principles (stated as invariants) are negative in that they indicate that certain undesirable events do not occur. A potential user of the system should also be interested in the positive aspects of the protection state, and how it is changed by invoking system operations. This is the subject of Section 8.6, which discusses the role of variant assertions.

We believe that the system can be proven correct with respect to the first two principles and the variant assertions, with only a few modifications in the design. The satisfaction of the first two principles and the variant assertions guarantees properties of security that are generally accepted by the "security community." That is, as we show later they rule out penetrations of the system and provide for acceptable change to the protection state of the system. Hence we will define the system to be secure if its implementation satisfies the first two principles and the variant assertions. It appears that achieving satisfaction of the last two principles may be extremely difficult, although little effort has been made thus far.

8.2 The Meaning of Security

One goal of this project has been to formulate the problem of proving the correctness of the operating system with respect to desired properties of security. As in the case of any proof of correctness, the credibility of the proof is bounded by how well the formal specifications characterize the intended behavior of the program.. In the literature there are numerous examples of program proofs that seem to be correct with respect to assertions that do not correspond to the goals of the program. For operating systems, as opposed to well-understood numerical programs, it is particularly difficult to state formally the desired behavioral properties, for example, those of a scheduler or a page manager.

We recognize that such a discrepancy between intent and assertions is indeed possible with regard to the security of the system. In particular, it is possible that any set of security assertions is incomplete. In order to avoid this undesirable event we have attempted to cast the security of the system in terms of a few believable assertions. In particular, four relatively simple principles are stated that are intended to prevent any possible security violation, along with a set of variant assertions that guarantee desired system behavior. It should be possible to prove that the specifications of each user-visible system function satisfy these principles. It is only necessary to handle the user-visible functions here; initially, a user will not be able to call nonvisible system functions immediately following login because of the initialization and authorization; second, a user will never obtain the right to call such functions during the course of execution, because of the principles discussed below.

Among the ways that a system might possibly be nonsecure, the most basic ones deal with the unauthorized modification or detection of information. Such violations are precluded by the following two principles:

P1: It is impossible for a user to alter information without authorization (Alteration Principle)

P2: It is impossible for a user to obtain information without authorization (Detection Principle)

The "information" in P1 and P2 can either belong to the system (e.g., passwords of all users) or can belong to an individual user (e.g., a segment or a directory). Both of these principles can be expressed as invariants. For example, P1 could be written as "Any information in the system will remain unchanged by any operations called by unauthorized users."

These two principles are discussed formally in the next section, but it is convenient here to justify them relative to our computational model. For these principles to be useful, it is necessary to understand and believe the meaning of user, information, alter, detect, and authorize.

The most simplistic view of a "user" is that it is the process running on behalf of some applications program. However, this definition is not adequate for characterizing the protection aspects of a user. For the purposes of this discussion, the most important property of a user is "his" access rights; the particular instructions that are being executed on his behalf are not important. At login, a process and an environment are created, and access rights are stored within the environment. Among the instructions that can be executed within this environment is "call" to an environment (or procedure) which gives control to the called environment. The rights stored within this new environment are a combination of those passed by the old environment and those that the called segment possessed prior to the call; the called environment still runs on behalf of the original process. The access rights that the calling environment possessed but did not pass are irrelevant to the execution of the called environment. Hence, a user for security purposes is viewed as an environment.

"Information" in our system is represented as the outputs of V-functions. An environment obtains (or detects) information by calling V-functions and having the output values delivered to the calling environment. Information can be altered only by calling an O-function or an OV-function.

"Authorization" requires somewhat more effort to define, but is intimately associated with possessing an appropriate capability. Let us assume that in order to perform any operation an environment must have (1) a capability to cause execution of the operation and (2) capabilities for the parameters of the operation. At login, an environment is created and initialized with capabilities to call some of the external system functions.

Also at login the environment is likely to be initialized with a capability for some system directory or process directory that itself might contain capabilities for language processors, library routines, etc., and possibly capabilities that can enable the environment to call V-function outputs shared with other environments. An environment e can obtain a capability that it never possessed only by calling a function that generates a brand-new capability, or by being passed the capability from some environment e1 that possesses it, or by having e1 place the capability into an object for which both e and e1 possess a capability.

The notion of an "object" is not fundamental for P1 and P2, but it is often more convenient and more intuitively meaningful to refer to a user as having access rights to an object rather than to V-functions. For our purpose here, an object associated with a module is defined as a set of V-function outputs of that module. Two distinct objects of a module have no V-functions in common. If one of the V-function outputs associated with an object returns a capability c, then we say that c is stored within the object. An environment is said to possess access rights to an object if the environment possesses all of the capabilities needed to call the O- and V-functions associated with the access rights. For example, if an environment possesses a segment capability s with access code {"read," "write"}, then it can call the O-function "write(s,i,j)" and "read(s,i′)". Thus the environment is said to possess "write" and "read" access rights to the segment identified by "get_uid(s)".

It is important at this point for the reader to become convinced that P1 and P2, if satisfied, rule out all abuses of the system that involve unauthorized modification of objects maintained by the system. They also rule out most user attempts to illegally obtain information from the system or from another user. The principles, in themselves, do not rule out all security violations, for example

(a) Those that arise from errors in implementations. Our concern in this chapter is entirely with proofs of specifications.

(b) The inferring of information from supposedly confined environments (see P4 below, and Section 8.5).

(c) Those that arise from users' programming errors.

The invariant principles P1 and P2 are fundamental to the protection of information in that they explicitly cover everything that a user can do to the information contained within an object, namely modify the information, or read values that represent the information. The formal statement of P1 in the next section guarantees that a user cannot call an O-function to alter the value of any V-function output unless he possesses the capability (or capabilities) required to call those V-functions. The only exception involves an object newly created by the O-function. The formal statement of P2 guarantees that the new values obtained by a V-function after an O-function call will not be dependent on V-function values for which the user does not possess a capability, unless (as above) the O-function creates a new object. These two principles, if proven to be satisfied by the system functions, guarantee that a user cannot "break the system." This can be seen as follows. In order to manipulate an object, a user must present a capability (or capabilities) for the object; P1 excludes any alteration without a capability. Thus a user cannot alter, say, a segment unless he possesses a capability c1 for that segment. Assuming that the user's environment is not initialized with such an unwarranted capability c1, then a user could only get such a capability by calling on some system V-function V which could return c1, where the user possesses a capability c2 for calling V. However, by P1 and P2 the V-function V will yield only the "information" c1 as a value if some user who possessed c1 called on some O-function that modifies V, presenting c1.

Thus if no user possessed cl initially, then no user can call on any function that will give him cl.

The remaining invariant principles, P3 (denial of service) and P4 (confinement) are discussed next, although this time in the opposite order.

The formal statement of the detection principle in the next section is carefully worded to exclude V-functions that do not have capabilities as arguments or V-function values associated with objects created as an effect of an O-function call. By calling such V-functions it is indeed possible for a user to detect some information belonging to the system or to communicate illegally with another user. Such behavior violates the Confinement Principle:

> P4: There shall be no inferring of protected information.

By "protected" information we mean objects for which the caller does not possess a capability. The principle can be stated as an invariant as follows: The information that user A has about some object for which A does not possess a capability (possibly belonging to the system) cannot increase by A calling any system function or any properly written user function.

As discussed in Section 8.5, it usually takes an "inference" process to obtain any protected information, since direct detection is ruled out by P2. Such inference can be accomplished by a combination of observing pertinent V-function outputs together with using a knowledge of the algorithm that generated the values of these V-functions. It appears that the only channels for communication in the present system will be via observing the time required for operations and the unique identifiers associated with capabilities. If the latter channel is considered harmful, it is possible to seal off all operations on unique identifiers except determining the equivalence of two arbitrary unique identifiers.

The other invariant principle is the Guaranteed Service Principle:

P3: There shall be no unauthorized denial of service.

This principle is intended to ensure that each user will receive some form of service from the system. That is, it is desired to prove, for example, that the scheduling and page replacement algorithms do not ignore any users. This principle has not been precisely formulated; in fact, difficulties are expected in the attempt. It is difficult to provide a precise formulation of service for a system which can serve a number of processes in a time-varying fashion. In particular, most contemporary time-sharing systems do not guarantee each user a fixed share of the cpu and main memory, but only a percentage of such resources averaged over some time frame. Thus it is difficult to even specify the service that a user should get in the absence of any unauthorized attempt to deny him service.

These four invariant principles are intended to exclude erroneous behavior on the part of inter-user interactions or user-system interactions. It is also desirable to guarantee correct behavior of the system. That is, it is useful to know what is the effect of carrying out some operation besides its not causing any security violation. In general it should be possible to state and prove properties about the operating system, e.g., that files are stored reliably. Since in this chapter we are only concerned with security properties, we wish to formulate some properties about a security state, and to assert how the protection state is modified by calling system functions. We achieve this by writing assertions for each system function that characterize the expected change in the protection state due to a call on the function. (These assertions are called variant assertions since they refer to expected variations in the state, as compared with the four invariant principles that refer to non-changes.)

Since the accessing of information is accomplished in the system by presenting capabilities to functions, it is natural to define the protection state of an environment as the set of capabilities accessible to the environment.

Two auxiliary functions are defined for purposes of concisely characterizing the protection state: one which defines the set of capabilities within an environment (at any instant), and the other which defines the total set of capabilities which an environment can obtain by calling system V-functions. The first function gives all of the capabilities that an environment has immediate access to, while the second function gives the capabilities that an environment can ultimately obtain, i.e., store in the environment, without being passed new capabilities from other environments. A typical way that an environment can augment its set of capabilities is to read a segment location where a capability has been previously stored.

The variant assertions are of importance since they define the meaning of the protection state, and characterize the positive effects of each system function on the protection state. A user of the system can apply the variant assertions in bounding the effect of giving away a capability to another environment.

The following comments serve mainly to contrast with other approaches the present approach to formalizing security. The reader who is not familiar with prior work may wish to ignore these points until he has completed the chapter.

- The results discussed here are preliminary. For example, more formal statements of the four principles are required in order to carry out "believable" proofs.

- The first two principles are stated as global assertions which can be proved from the system specifications by the induction method discussed in Chapter 3.

- There is a strong relationship between the four principles and the variant assertions, and the degree to which the system is certified. If just the first two principles are considered, then it is guaranteed that the system is protected from abuse and that a user can have some confidence that other users cannot get access to his information, except through low bandwidth channels. The satisfaction of these negative principles is the main goal of security kernels. We feel that these two principles are not completely adequate in themselves, since, as a system goal, they do not guarantee that the system will carry out any useful work. The remaining two principles, P3 and P4, guarantee much stronger properties but are harder to formulate and prove. The third principle, P3, guarantees that each user will get a "fair" share of the system. The variant assertions formally specify the positive effect of an operation on the protection state. For example, they describe the effect of a user giving a capability to an object.

- The formal statements of the alteration and detection principles can be viewed as an axiomatization of the properties of capabilities in a formally specified system. The statements define what an object is relative to V-function values, and give properties that capabilities should have in order to provide protection for their associated objects against abuse by O-function calls. A reader who believes the formal statements of these principles and the model of capability interpretation and generation should believe that the system provides protection of objects.

- It is useful to contrast the statement of the principles with the standard access matrix approach to describing protection (Graham and Denning [72]). The access matrix is a data structure storing the access rights of users to objects, and is basically a static formulation of protection. The access matrix was originally suggested as a system table for storing the protection state of the system. This idea in its "pure" form has since been rejected because the access to the table contents is inefficient due to the sheer size of the table. The matrix is not adequate for stating formal properties of protection because it is too cumbersome. Instead we represent access in terms of functions, which can be interpreted as defining an access matrix when all elements in the domain and range are considered.

- The effort here is concerned with stating properties of the operating system design, i.e., of the specifications. There still remains the (substantial) effort to prove that these properties are valid, and to prove that the implementation is correct with respect to the specifications.

- The proof of the security afforded by the design is contingent on the proper initialization of each user's environment of login. If the initial state does not allow a user to violate the security rules, then there is no function that can be called (by the user or by other users) that will produce a state in which security can be violated.

- There is no fundamental distinction between accidental and malicious violation of security. For example, an error in a scheduler, however unintentional, might be exploited by a user to deny service to other users.

8.3 The First Two Security Principles

This section gives precise statements of the alteration and detection principles. Because the statement of these principles is abstract, they will be motivated by considering several examples of actions that are clearly violations of security.

Suppose that c is a capability corresponding to a segment and that "read(c,i)" is defined. Suppose further that c' is also a segment capability and that "read(c',j)" is defined.

Security Violation 1. Suppose that, contrary to fact, there existed an O-function in the operating system called "illegal_write1," which had the effect that when "illegal_write1(c',j,x)" is called, the value of "read(c',j)" is set equal to c for $c \neq c'$ and $c \neq x$. Clearly, "illegal_write1" has violated security if the environment that executed the call of "illegal_write1" did not have c within its environment, because the environment may now obtain the capability c merely by calling "read(c',j)".

Security Violation 2. Suppose that, contrary to fact, there existed an operation in the operating system called "illegal_write2," which had the effect that when "illegal_write2(c',i,y)" is called, then "read(c,i)" is set to 5 for $c \neq c'$ and $c \neq y$. Clearly, we would reject the idea that such an operating system were secure, because it contains a classic example of altering "someone else's" data.

Security Violation 3. Finally, suppose that, contrary to fact, there existed an operation called "illegal_write3" which had the effect that when "illegal_write3(c,i,j)" is called, then "read(c',i) is set to 1 if "read(c',i)" is 0 for $c \neq c'$. Such an operation would also violate security because, even though no illegal alteration was performed (the author of the action does have c), the outcome of the operation depended upon something not accessible from the capability passed as an argument to the operation.

As seen below, security violations 1 and 3 violate the detection principle, and violation 2 violates the alteration principle. One aspect of security amounts to forbidding actions such as those described by security violations 1, 2, and 3. The difficult part of the definition of security is in accurately generalizing from such examples to a clear principle.

At any moment, there exists a set of capabilities called NEW, which is the set of all capabilities that are not and never have been either the values of any user-accessible V-function, or members of an argument list of a V-function, or the value of any OV-function call. This definition of NEW is required in the statement of the alteration principle. Wherever the system returns a capability c for a newly created object, c must be a member of NEW. After the return from the call that returns the capability c, NEW is diminished by c.

The Alteration Principle

Given an O-function call "O(a1,a2,...)", suppose that the V-function value "ans = V(b1,b2,...)" after the O-function call is different from the value of the same function with the same arguments before the call. Then any member x of {ans,b1,b2,...} that is a capability must be either a member of {a1,a2,...} or a member of the set NEW before the call. After the calls, all such x cannot be members of the resultant set NEW. The restriction for V-functions as a result of the OV-function call "ans = OV(a_1,a_2,...)" must be the same as for an O-function call. Also the effect of an O- or OV-function call on the value returned by an OV-function must be the same as that for the O- or OV-function call on the value returned by V-functions.

It is important to understand precisely what the alteration principle guarantees relative to protection. If a system O- or OV-function modifies any V-function that requires a capability c to call, then c must be either a new capability or a capability that the caller of the O-function must himself have presented. Thus a user cannot call any system function that requires a capability to modify any information, unless he has a capability for the V-functions associated with that information. Furthermore, if a V-function output is modified by an O-function to a capability value c1, then the caller of the O-function must have possessed and presented c1. This guarantees that if a V-function returns to a caller a capability c1, then either the caller possessed c1 to begin with, or received rights to the object that holds the capability c1 from a user who possessed c1.

To show that it is possible to prove that specifications can satisfy the alteration principle, consider the O-function "write" and the V-function "read". "write(c,j,w)" causes the values of "read(c,j)" to change. That is acceptable under the alteration principle because the capability c, which occurs as an argument to "read" occurs as an argument to "write," and since the new value of "read(c,j)" does occur as one of the arguments of "write". We have not yet attempted to prove that the specifications for all functions satisfy the alteration principle. However, we feel that only a straightforward invocation of the specifications is needed to accomplish this verification.

This should give to the reader a rough understanding of what is a reasonable effect for an O-function to have. To illustrate how an O-function might violate the alteration principle, suppose that the effect of "write (c,j,w)" were to alter "read(c′,j)", where c′ is a different segment capability from c. Then such an instruction would violate the alteration principle.

Prohibiting the undesirable alteration of the state of the machine, however, is only one aspect of the idea of security. Another aspect that can be formulated is the exclusion of the detection of parts of the system or of other users that the environment issuing the instruction is not authorized to detect.

The Detection Principle

> Given an O-function call "O(a1,a2,...)", suppose that the state of the abstract machine changes as a result of the call. If the state change that results is dependent on the prior value (before the call) of some system V-function V (b1,b2,...), then all of the bi that are capabilities are in {a1,a2,...}, or are members of NEW before the call and are not members of NEW after the call. A similar statement applies to an OV-function call "ans = OV (a1,a2,...)" as to "O(a1,a2,...)".

The detection principle states simply that any information modified as a result of an O-function call can depend only on information for which the caller presented capabilities, on newly created information, or on information which does not require a capability. Note that the detection principle is not concerned with a V-function value being modified to return a capability. The alteration principle gives precise conditions under which a V-function can be so modified.

Another viewpoint of the detection principle is related to the limitations it places on unauthorized communication between users. The primary channel for user A sending information to user B in the system is by user A changing the value of some V-function that user B can call. The detection principle rules out such communication via a V-function that requires a capability as an argument unless A and B have a capability in common. The detection principle does not apply to V-functions that do not include a capability as an argument. However, there are no such functions visible to users of the operating system. The detection principle also does not cover subtle issues of the confinement problem (see also Section 8.5).

In summary, the detection and alteration principles certainly guarantee that the system will not give away crucial information, for example, capabilities for its tables, but a user might be able to infer some system information (for example, the average workload), although it should always be minimal and noncritical.

8.4 Extension for Access Codes, Revocation, Call, and Return

The alteration and detection principles as stated above are credible statements that capture the intuitive meaning of unauthorized alteration and detection of information. The principles can be viewed as an axiomatization of one aspect of capabilities--namely their role in uniquely identifying objects. For conciseness, the previous section does not address the role of the access code portion of the capability or the role of revocation. This section presents slightly more detailed statements of the two principles, to reflect these additional roles. It also shows that the present specifications for "call" and "return" and a few other functions do not satisfy the principles, but that the specifications can be revised to conform thereto.

8.4.1 The Alteration Principle Revised to Accommodate Access Codes

If the unique identifier (uid) portion of a capability uniquely identifies an object, then the access code portion of the capability identifies the set of operations that the user of the capability may perform on the object. For the present discussion, two capabilities which have different uid's are considered to be completely unrelated. That is, the operation "write(c,j,w)" and "write(cl,j,w)", where the uid of c is different from the uid of cl, should produce entirely different effects--in particular, the two operations correspond to distinct segments. However, if the uid of c is equal to the uid of cl, and the access vector of c is {"write," "enter"} and the access vector of cl is {"write"}, then the operations "write(c,j,w)" and "write(cl,j,w)" should have the same effect. If, on the other hand, the access vector of cl is {"read," "enter"}, then the operation "write(cl,d,w)" will produce an error. Similarly, if c2 and c3 have the same uid but the access vector of c2 is {"read"} and the access vector of c3 is {"read," "write"}, then the operations "read(c2,j)" and "read(c3,j") will produce the same effect.

We can now state the Access Right Principle:

> If a V-function is defined (i.e., returns a value) for some argument list, and if one of the arguments is replaced by a capability with the same unique_id but a bigger set of access bits, then the V-function returns the same value. Similarly, consider an O-function or OV-function "f" whose execution does not cause an error (i.e., has an effect). Suppose c is a capability in the argument list for function "f", and is also a member of the argument list of some V-function that is modified by the execution of f. Suppose c′ is a capability with the same unique_id as c, but with a larger set of access bits. Then the same effect could be achieved by replacing c with c′ as an argument to "f".

Note that the statement of the Access Right Principle for O- and OV-functions would not be correct if it said that any capability in the argument list of the O-function is replaced by a capability with the same unique_id but a bigger set of access bits. For example, consider the operation "write(c,j,w)" where w is itself a capability. Clearly, replacing w by a capability with a larger set of access bits could produce a different effect on the segment word being written.

The Access Right Principle is merely a formal restatement of our intuitive understanding of how uid's and bits in the access code are interpreted by a type manager for accessing an object. If a capability with a "1" in a particular access bit position is required for an operation, then the operation will be carried out successfully independent of the values in other access bit positions. The alteration principle is now restated to account for the role of access bits. (Note all members of the set NEW are assumed to possess distinct unique_id's.)

The Alteration Principle (with Access Codes)

> Given an O-function call "O(a1,a2,...)", suppose that the V-function value "ans = V(b1,b2,...)" after the O-function call is different from the value of the same function with the same arguments before the call. Then any member x of {ans,b1,b2,...} that is a capability is a member of the set NEW before the O-function call, or it has a unique_id portion that is identical to that of a capability of {a1,a2,...}. After the call, all such x cannot be members of the resultant set NEW. The effect on the V-functions as a result of an OV-function call "ans = OV(a_1,a_2,...)" must be the same as for an O-function call.

It should be clear that the revised statement of the alteration principle states that the unique_id portion of a capability is adequate for bounding the V-function values that are allowed to be modified as a result of an O-function call. As an example, the operation "write (c,j,w)" where the access vector of c is {"write"} causes the value of "read(c1,j)" to become w, where c1 has the same unique_id as c, but has an access vector {"read"}. This does not violate the revised alteration principle, even though c $\neq$ c1.

The detection principle must also be slightly revised in order to account for the unique rule of the unique_id in bounding the permitted values that V-functions can attain.

The Detection Principle (with Access Codes)

> Given an O-function call "O(a1,a2,...)", suppose that the state of the abstract machine changes as a result of the call. If the state change that results is dependent on the prior value (before the call) of some system V-function V(b1,b2,...), then all of the bi that are capabilities must have a unique_id portion that is identical to that of a capability in {a1,a2,...}, or must be members of NEW before the call and are not members of NEW after the call. A similar statement applies to OV-function calls.

To see the application of the revised statement, consider an O-function which could have been in the system: "writes(c,j,c1,j1)" which stores the contents of location j1 of segment c1 into location j of segment c. Assuming the capability c1 has access code {"read"}, then the effect of the call is to have read(c,j) = 'read'(c1,j1). If there is another capability c2 in the system with the same unique_id as c1, but access code {"read," "enter"}, then the modified value of "read(c,j)" appears also to depend on "read(c2,j1)". Since the unique_id's of c1 and c2 are equal, the above revised statement of the detection principle is not violated in this case.

8.4.2 The Significance of Revocable Capabilities and "Revoke"

Until the recent work of Redell and Fabry [74], it was a common criticism of capability-based operating systems that once a user had possession of a capability, there existed no convenient way to later deny that user access to the values of V-functions accessible via that capability. However, based on work of Redell and Fabry, the present operating system incorporates a convenient mechanism for "provisionally" giving a capability to an environment; the provision is that one can use the O-function "revoke" to invalidate the capability given away. The existence of revocation, while convenient, does slightly complicate the statement of the security principles.

A user who wishes to give away provisional access rights associated with the capability c initially calls the function "cr = create_revocable_cap(c)". Externally the capabilities c and cr appear to be virtually identical, in that if c is accepted as an argument to an O- or V-function, then cr will also be accepted. The unique_id's of c and cr are different, but the interpretation by system functions is based on the value of "chase_uid", and since chase_uid(c) = chase_uid(cr), the capabilities seem to be interchangeable.

However, besides the difference in uid values, there are other differences between c and cr. The access code of the capability cr is augmented with the ability "revoke." Assuming that the original possessor of cr does not wish to pass the "revoke" rights to other users, he can call the function "c1 = restrict_access(cr,i)" where i is the access code of c. At this point, c1 and c appear to be virtually identical for access to objects. Until "revoke(cr)" is performed, the capabilities c, c1, and cr are equal as far as can be determined by the user. However, once "revoke(cr)" is performed, neither cr nor c1 can serve successfully as the argument to any V-function (i.e., any V-function value will forever become undefined for any argument list which contains cr or c1).

Let us now consider the effect of revocable capabilities on the alteration principle. The act of calling "revoke(cr)" has the effect of modifying all V-functions that have the capabilities cr and cl as arguments. The effect of the modification is to cause these calls to return an error condition. This does not violate the revised alteration principle, since cr is a member of the argument list of "revoke" and has the same uid as the capability arguments of the affected V-functions. However, consider the case of two or more levels of revocable capabilities. Let cr result from calling "create_revocable_cap(c)", and let crl result from calling "create_revocable_cap(cr)". Then the act of calling "revoke(cr)" modifies all V-functions that have crl as an argument (even though crl is not an argument to "revoke(cr)") and thus violates the alteration principle since the uid of crl is not the same as the uid of cr. Besides "revoke," other O-function specifications do not satisfy the alteration principle assuming revocable capabilities as arguments. An O-function call will change V-function values for all capability arguments that have identical values of "chase_uid". Thus a restatement of the alteration principle can be formulated to account for revocable capabilities, also subsuming access codes.

The following restatements of the alteration principle and the access right principle are most easily performed by introducing the concept of "R_EQUAL(n,m)". Two unique_id's n and m are R_EQUAL if, in the language of LEVEL 4 of the operating system design, last(linktuple(n)) = last(linktuple(m)). An interpretation of this statement is that the uid's n and m are on the same indirection tree, where each call on "cr′ = create_revocable_cap(c′)" introduces another level of indirection emanating from the uid associated with c′. For the capabilities cl, c2 associated with n, m, an equivalent interpretation of R_EQUAL(n,m) is that chase_uid(cl) = chase_uid(c2).

The Alteration Principle (with Revocable Capabilities)

Given an O-function call "O(a1,a2,...)", suppose that the V-function value "ans = V(b1,b2,...)" after the O-function call is different from the value of the same function with the same arguments before the call. Then any member x of the set {b1,b2,...} that is a capability has a unique_id that is R_EQUAL to the unique_id of some member of the set {a1,a2,...}, or is a member of the set NEW before the O-function call occurred. After the call, all such x cannot be members of the resultant set NEW. If "ans" is a capability, then it is a member of the set {a1,a2,...}. Given an OV-function call "ans = OV(a1,a2,...)", the same properties are required as in the case of an O-function call.

The access right principle can also be restated to account for revocable capabilities. We assume that a V-function will become "undefined" if one of its arguments that is a revocable capability is revoked by another user.

The Access Right Principle (Modified for Revocable Capabilities

If a V-function is defined (i.e., returns a value) for some argument list, and one of the capability arguments c is replaced by a capability c1 with an R_EQUAL unique_id and a bigger set of access bits, then the V-function, if it is defined with c1 as an argument, returns the same value. Similarly, consider an O-function or OV-function "f" whose execution does not cause an error. Suppose c is a capability in the argument list for "f"

> and is also a member of the argument list of some V-function that is modified by the execution of "f". Suppose c' is a capability with a unique_id R_EQUAL to c, but a bigger set of access bits. Then the same effect could be achieved by replacing c with c' as an argument to "f".

At present there are a few user-visible O-functions and OV-functions that do not satisfy the above alteration principle. To show the current lack of conformity in the case of "revoke", consider any V-function that requires two (or more) capabilities in its argument list, e.g., the level 5 function [c] = impl_cap(c1,c3). If c1 is a revocable capability, then the O-function call "revoke(c1)" clearly modifies the value of "impl_cap(c1,c3)". This is in violation of the alteration principle, since chase_uid(c3) is not in the argument list of "revoke(ci)". This violation of the alteration principle does not appear to be significant, since intuitively "revoke" does perform as intended. More effort is needed to develop a revised statement of the alteration principle that circumvents this conflict with revocation.

There is an additional lack of conformity to the alteration principle in the case of "revoke". The purpose of calling "revoke(c)" is to cause all V-functions to become undefined that have as an argument any revocable capability ci that is a result of calling the function "create_revocable_cap" with an argument c, or with some ci that is itself a revocable capability corresponding to c. The state change that results from calling "revoke(c)" violates the statement of the alteration principle in that the V-functions that are affected do not necessarily have c as an argument. (Note that the concept of R_EQUAL unique_id's does not apply here since "revoke(c)" will not have any effect on a capability c1 if c is a revocable version of c1.) It appears that a solution is to define a new relation INDIRECT(c2,c3) among the capabilities c2 and c3 that reflects c3 being a revocable version of c2. Then the alteration principle can be modified to account for the effect of revoke(c2) on all V-functions that take as arguments capability cj such that INDIRECT(c2,cj) is true.

At first glance, it appears that the statement of the detection principle is violated by the function "revoke(c)", since the specifications indicate that the resultant state change is dependent on the values of the hidden V-functions "linktuple" and "revocable_set", and that the capability argument c of "revoke" is not a capability for "linktuple" or "revocable_set". However, the unique_id's contained within these V-functions are associated with capabilities {ci} that satisfy the relation INDIRECT(c,ci). It thus appears that a simple augmentation of the detection principle results in its being satisfied by the function "revoke".

8.4.3 The Significance of "call" and "return"

One reason for having the functions "call" and "return" is to solve the following problem. Assume a pair of users A and B such that B has written a program and A wishes to use the program, but certain constraints are desired:

1. A does not want the program to have any way of obtaining any of A's capabilities except those particular capabilities that A is willing to pass to the program.
2. B does not want A to know anything about the details of the program and does not want A to be able to obtain the capabilities that may be embedded in the program.

These two objectives are achieved by the use of "call" and "return", and by the "enter" access right for segments. When B creates his program, he writes it into a segment and he gives to A a capability for the segment with only the "enter" ability. There is only one way that A can use such a capability, and that is to use "call"; "call" creates a new environment to which A does not have any original access, and begins the execution of the program. A is allowed to pass to the program certain arguments, among which may be capabilities.

Among the capabilities that are passed is a capability to return back to A when B is finished. Thus the arguments of "call" are (a,b,[p]) where a is the entry capability, b is the return capability, and [p] is the tuple of passed capabilities. (The specifications of level 10 differ slightly in format from that discussed here, for descriptive simplicity.) The capability "a" points to the address where B's program is to begin execution--typically the beginning of the segment. The capability "b" points to the address in the calling program where execution is to continue when control is returned.

When B's program is finished processing A's data, control is to be returned to the original calling environment. At this time B's program executes the instruction "return(b,[p′])" where [p′] is a tuple of capabilities to be passed back to the original environment.

The function "call(a,b,[p])" does not violate the alteration principle by causing the creation of a new environment associated with a unique capability contained within NEW. The fact that the capabilities [p] are stored within the new environment is again not a violation of the principles. However, the fact that the new environment begins executing from the segment that contains B's program is apparently a violation of the detection principle. This violation arises from A not having any capability for the contents of the segment, in particular the starting address. The new state of the system, in particular the program counter, is dependent on functions for which A does not possess a capability. However, it appears that the detection principle will be satisfied under the proper interpretation of the "enter" capability "a". In particular, the "enter" capability can be interpreted as giving the caller certain access rights to the new environment, namely to cause it to commence executing the program associated with "a" at the indicated entry point.

The function "return (b,[p$'$])" causes a modification of the original calling environment, in particular by augmenting that environment with the elements of [p$'$]. This is not a violation of the alteration principle if we interpret the capability "b" as giving the returning program "append" rights to the original environment at return time.

The detection principle is in question for "return" since the value of the program counter is dependent on the instruction address just beyond the calling instruction in the calling program, a location to which the called environment has no apparent capability. However, again under the proper interpretations of the "return" capability "b" there is no violation.

In general, work is still necessary to formalize the notions of "enter" and "return" capabilities so as to lead to conformity with the two principles. However, there do not appear to be any intrinsically difficult problems.

8.5 Discussion of the Confinement Principle (P4)

A particular problem that all allegedly secure operating systems should be able to solve efficiently is the realization of a memoryless subsystem. User A wishes to have his data processed by a program written by B. However, A does not wish B to have any access to A's data after B has completed the processing. If A does not trust B's program to conform to this desire, then the normal method of A calling B does not work. The only solution is to have execution take place in a special memoryless environment e. The primary feature of e is that it will not commence processing if it is passed a capability for permanent storage, e.g., with "write" ability for a segment.

Thus at first glance it appears that the problem is easily solved, although there remain some unresolved issues relating to efficiency and nested calls within the memoryless environment e. It does not seem that B can retain any of A's data or give the data to some other cooperating user C (for later retrieval by B) since B, within the environment e, is guaranteed not to share a capability with any other user C. However, there are ways for two users to communicate other than via some object for which each possesses capabilities.

Lampson [73] describes a possible communication channel whereby information is transmitted by B opening a particular file and C testing the status of the file. We do not envision the existence of this particular channel in our system since any operation on a file will require a capability. However, we do anticipate that two users, B and C, could communicate by observing the time required for operations. User B could invoke an operation that is likely to require a long time to complete, and possibly cause an operation of C's to get delayed. If B and C are cooperating, then it is possible for B to send information to C via this channel. (Even if the system clock is not accessible, a small amount of information can be transmitted if C has other means for observing how long he is delayed, e.g., his wristwatch.)

We are aware of the existence of one other channel by which two users who are not supposed to communicate can possibly communicate--albeit at a low rate. Just as all users may in some ways be able to share the system clock, all users in our system share the mechanism that creates unique_id's. The algorithm that generates a new value of unique_id whenever "create_cap" is called has not been specified, but among the possibilities for the set NEW are: (1) an ordered sequence of integers, (2) values derived from the system clock, (3) a more complex encoding. In any event, since a fixed value is not returned, the unique_id creation mechanism is a source of information. It is possible for two users to communicate, if they have some knowledge of the algorithm.

We believe that these two channels are the only ones that will be present in the final system. The channel capacity for such communication will probably be low, but it is recommended that the rate be studied from an information theoretic viewpoint for better understanding of the problem.

8.6 Variant Security Assertions

The previous discussion is concerned with global assertions at the user-visible interface, describing the nonoccurrence of undesirable events. This is only part of the requirement for security. It is also important to know precisely how each operation affects security, e.g., writing a capability into a segment. The approach here is to define a security state in terms of auxiliary functions, and then to state the effects of system functions on the "value" of these auxiliary functions. These effects are expressed as variant assertions. Ultimately such assertions will exist for each of the functions of the system; a few illustrations are given here. A user of the system should be able to read these assertions, and easily be convinced of the proper behavior of the system functions with regard to their effect on the security state.

The following auxiliary functions serve to define the protection state of the system:

(1) object: $[c] \rightarrow \{\text{TRUE, FALSE}\}$. This function indicates whether the given tuple $[c]$ of capabilities denotes an object.

(2) imm_access: $[c] \rightarrow \{c'\}$. This function has as a value the set of capabilities $\{c'\}$ contained in the object denoted by $[c]$. Some system objects, e.g., segments and directories, can serve as repositories for capabilities.

(3) iter_access: $\{c\} \rightarrow \{c'\}$. For a set of capabilities $\{c\}$, this function gives the set of capabilities $\{c'\}$ that are obtainable by calling system V-functions with one (or more) capabilities contained in $\{c\}$. Initially $\{c'\} = \{c\}$. Each such V-function call may expand $\{c'\}$. Repeated calls are made with the expanded capability set, until no further augmentation is possible.

(4) env_iter_access: $e \rightarrow \{c\}$. This function has as a value the set of capabilities $\{c\}$ that can be iteratively obtained from the set of capabilities initially in the environment denoted by e.

The effects of various system functions on these four auxiliary functions are now considered. The only effects that need be described are those on the two functions "object" and "imm_access". The effects on "iter_access" and "env_iter_access" can be derived from the other two functions as follows:

```
LET $x = \{[c'] \mid object([c']) \wedge (tuple\_set([c']) \subseteq \{c\})\}$;

  /Comment: The function "tuple_set([c'])" transforms a
   tuple [c'] into a set {c'}; it eliminates all replicas
   of a capability and disregards the order of the elements.
   Thus x is the set of all capability tuples that denote
   objects in [c]./

LET $y = \{c \mid \exists [c'] (c \in imm\_access([c']) \wedge ([c'] \in x))\}$

  /Comment: Thus y is the set of all capabilities that are
   contained in objects pointed to by a tuple of capabilities
   [c'] contained in x./

iter_access({c}) =

   IF $\{c\} = y \cup \{c\}$ THEN $\{c\}$
   ELSE iter_access $(y \cup \{c\})$;

env_iter_access(e) =

   IF get_type(e) = "environment" THEN iter_access(imm_access([e])
   ELSE UNDEFINED
```

For the present, access bits and revocation are ignored. The reduction of access that occurs during a call is also omitted.

All V-function calls of the system change the environment that calls them. Let e be the calling environment. Then the variant assertion for the call of the system V-function V(c1,c2,...) that returns a capability c is as follows:

EFFECT: imm_access([e]) $\subseteq$ 'imm_access'([e]) $\cup$ c.

Let us now consider the more interesting case of variant assertions for O-functions.

insert_impl_cap(c,t,c1) /level 5/

EFFECT: imm_access([c,t]) $\equiv$ 'imm_access'([c,t]) $\cup$ c1

This assertion states that the object of extended type, maintained by level 5, defined by [c,t] is augmented with the implementation capability c1.

insert_entry(d,n,c) /level 6/

EFFECT: imm_access([d]) $\equiv$ 'imm_access'([d]) $\cup$ c

move_entry(d,n,d1,n1) /level 6/

EFFECT: imm_access([d1]) $\equiv$ 'imm_access'([d1]) $\cup$ 'get_cap'(d,n);

imm_access([d]) $\subseteq$ 'imm_access'([d])

Object creation is accomplished by OV-functions and by "call." These operations do not add any capabilities to the objects, except to place a capability for the new object in the calling environment. Let e be the calling environment in the following security assertions:

c = create_object(t) /level 5/

EFFECT: imm_access([e]) $\subseteq$ 'imm_access'([e]) $\cup$ c;

object([c,t]) = TRUE

s = create_segment /level 4/

EFFECT: $\text{imm\_access}([e]) \subseteq \text{'imm\_access'}([e]) \cup s$;

object([s]) = TRUE

d = create_directory /level 6/

EFFECT: $\text{imm\_access}([e]) \subseteq \text{'imm\_access'}([e]) \cup d$;

object([d]) = TRUF

Each of the above is an OV-function, in which cases the returned capability could overwrite a prior capability in the environment. Thus "$\subseteq$" is used instead of "$\equiv$".

$\text{call}(c, c_1, c_2, \ldots, c_n)$ /level 10/

PURPOSE: c is the procedure segment and

$c_1, \ldots, c_n$ are the parameters passed on the stack

EFFECT: CHOOSE new $e' \mid$

$\text{object}(e') = \text{TRUE}$;

$\text{imm\_access}([e']) \equiv \text{'imm\_access'}([c]) \cup \{c_1, \ldots, c_n\}$;

$\text{return\_env}(e') = e$; /level 10/

Note that "return_env" is a V-function of level 10.

Deleting an object destroys the object, but does not increase access; thus the effects are not interesting.

The function "return" destroys an environment (the environment calling "return") and passes capabilities back to the return environment.

$\text{return}(c_1, c_2, \ldots, c_n)$

EFFECT: object(e) = FALSE;

imm-access(e) = UNDEFINED;

LET $e' = \text{'return\_env'}(e)$; /level 10/

$\text{imm\_acc}([e']) \subseteq \text{'imm\_acc'}([e']) \cup \{c_1, \ldots, c_n\}$

Many O-functions have no effect on the security state (e.g., wait, signal, enter_monitor, exit_monitor). Further work is required on variant assertions to handle access bits and revocable capabilities, and to produce a comprehensive list of variant assertions for all the functions in the system.

8.7 Conclusions

In this chapter we have made a preliminary attempt to define some statements that characterize what it means for a system to be secure. It appears that these statements are satisfied by most of the functions of the system. The existence of a few exceptions does not mean that the operating system is insecure, but that either the specifications or the security statements will have to be slightly modified.

The security statements are organized as five classes of statements, which if all satisfied mean that the system is totally secure. For the first two classes we have defined two principles which relate to the alteration and detection of information that can be accessed only via capabilities. If these two principles are satisfied, all such information is guaranteed to be protected. We expect that the system can be proven correct with respect to these principles.

The third principle relates to the system being secure with respect to providing service to users. Until we formulate a model of service in a many-process system, we will not be able to guarantee that there is no unauthorized denial of service. The fourth principle relates to leakage of information from one user to another--loss of confinement. The confinement problem is made difficult whenever system resources are available to all users. In our system, the only channels that seem to permit leakage are the unique_id creation mechanism and the observation of the time required for an operation. Although leakage can take place, it appears that the bandwidth of such leakage can be determined to be extremely low.

The fifth class relates to ensuring that positive security behavior is exhibited by the system. A set of assertions has been developed that describes the changes in the security state for each system function. Since the third and fourth principles do not relate to the modification or detection of crucial information in system objects, we say the system is secure if the first two principles and the variant assertions are satisfied.

The proof of security when completed does not in itself guarantee absolute protection. The security of the system is dependent on the proper initialization of environments created at login. Also a user's programs can be in error and thus unintentionally give away capabilities. However, since the system is completely specified, a user can always determine the effect of giving away a capability to another user.

Chapter 9

MONITORING OF SECURITY AND PERFORMANCE

Monitoring in general has two basic aspects, concerning security and performance. Although these two aspects are related in considering denial of service, they are otherwise rather distinct, and thus they are treated separately here. We consider monitoring of system alterations, monitoring of performance, monitoring of security in normal operation, monitoring of security during recovery (if necessary), and monitoring of denial of service. Monitoring of performance is related to accounting, resource allocation, and charging. These relations are also discussed.

Monitoring of security has a role in our system similar to that in most other systems. Traditional auditing of system usage is expected. However, those properties of the system that have been formally proven may not require as elaborate monitoring facilities as previously. In general, monitoring is greatly simplified by the existence of any assertions that have in fact been proven, and by the explicit nature of what is visible at each system level. As more and more proofs become available in a given system, corresponding monitoring functions may tend to be simplified.

This chapter is divided into two parts, the first being generally applicable to systems designed according to our methodology (Section 9.1), and the second being specifically relevant to the system described in this report (Section 9.2).

9.1 General Considerations for Monitoring Hierarchical Systems

Monitoring in a hierarchical system can be distributed and done at levels appropriate to the needs for detailed information. There are two basic modes in which monitoring may occur: monitoring of properties of the system as a whole, and monitoring of properties of individual processes (or of usage of specific users). Either mode may take place asynchronously by processes devoted to monitoring, or may be done interactively on request by authorized users. Individuals may monitor their own processes and in certain cases may also monitor other users (e.g., users subsidiary to them in a collaborative environment). In general, monitoring takes place at command level, although in some cases lower-level monitoring primitives may be used. Monitoring operations are in no essential way different from other operations in the system, and are subject to normal access via capabilities. Thus the ability to use monitoring operations may be selectively controlled, as desired. In essence, these operations involve obtaining the states of lower levels (i.e., using lower-level V-functions), but do not permit any state changes. In this way, proof that security is maintained despite monitoring becomes extremely simple in a system structured and specified according to our methodology. Note that certain state information may have to be made visible external to a level, in order to implement monitoring. However, such information is visible only in an abstracted form and then only to higher-level monitoring functions. Thus at command level, each unit of abstracted state information is available just to certain users (e.g., the security officers in the case of security monitoring).

Whether self-imposed or externally imposed, there are three types of monitoring:

(M1) monitoring as a part of the implementation of a function F being monitored, e.g., by including monitoring arguments

Chapter 9

MONITORING OF SECURITY AND PERFORMANCE

Monitoring in general has two basic aspects, concerning security and performance. Although these two aspects are related in considering denial of service, they are otherwise rather distinct, and thus they are treated separately here. We consider monitoring of system alterations, monitoring of performance, monitoring of security in normal operation, monitoring of security during recovery (if necessary), and monitoring of denial of service. Monitoring of performance is related to accounting, resource allocation, and charging. These relations are also discussed.

Monitoring of security has a role in our system similar to that in most other systems. Traditional auditing of system usage is expected. However, those properties of the system that have been formally proven may not require as elaborate monitoring facilities as previously. In general, monitoring is greatly simplified by the existence of any assertions that have in fact been proven, and by the explicit nature of what is visible at each system level. As more and more proofs become available in a given system, corresponding monitoring functions may tend to be simplified.

This chapter is divided into two parts, the first being generally applicable to systems designed according to our methodology (Section 9.1), and the second being specifically relevant to the system described in this report (Section 9.2).

in the call to and return from the function, or by creating new effects to the function (explicitly, rather than as side effects).

(M2) monitoring by enforcing the (synchronous) interposition of a monitoring operation M upon each invocation (or upon selected invocations) of a function F to be monitored. The interposition of M does not alter the implementation of the function F, but may examine the arguments of the call and the arguments of the return. One simple way such interposition may be achieved is by introducing monitoring functions as exception-handling routines, and explicitly triggering the corresponding exception conditions as required. Furthermore, a particular monitoring operation may be allowed to gain control on the occurrence of *any* exception whatsoever. This monitor could then be a unified and combined first-level handler for recovery, security monitoring and performance monitoring.

(M3) monitoring by a separate monitoring operation, typically asynchronous to the function(s) being monitored, whether external to or within any process(es) being monitored.

These three cases are summarized in Table 9.1.

9.1.1 Monitoring of Security for System Alterations

In general, it is assumed that, prior to their installation, all would-be system changes are proven to be consistent with the desired properties of the existing system. It is further assumed that there is a rigorous, and rigidly enforced, protocol on the part of the system and its administrators, guaranteeing that each would-be alteration has been proved or otherwise deemed worthy of installation, that the version

Table 9.1

SUMMARY OF THE THREE TYPES OF MONITORING OPERATIONS M WITH RESPECT TO A FUNCTION OR SET OF FUNCTIONS F

Type	With Respect to F	With Respect to "call F" and "return"
M1	Internal to F Usually visible to F*	Internal to "call F" and "return" Synchronous with F
M2	External to F Invisible to F	Internal to "call F" and "return" Synchronous with F
M3	External to F Invisible to F	External to "call F" and "return" Usually asynchronous with F†

*M1 may be invisible to F's source code via an interpreter or a compiler.

†M3 may be synchronous with F via external locking.

installed is in fact the proved version (rather than a forgery), and that any necessary reinitialization of the system to accommodate the alteration is itself correct. This can be achieved by using special directories or directory-like objects for system alterations, and by doing all alterations by a single command (e.g., "alter_system"), accessible only to the security-officer-for-alterations. (Note that the capability mechanism makes it possible to prevent unauthorized alterations, and to prove that they are impossible.) In some cases,of course (e.g., where the lower levels are themselves being changed), system alterations cannot be made while the system is itself in control. (See Chapter 10 for a discussion of hierarchical reinitialization during and after alteration.) In any event, system changes that affect all users should never be installed while the system is available to normal users.

In general, it is desirable that the mechanisms for effecting system alterations be uniform, irrespective of whether they are done in a no-user bootstrap-generated system, in a one-user restricted-access system (the security officer alone), or in normal operation. Thus the role of hierarchical alteration should be closely related to hierarchical initialization.

Note that changes by a user affecting only himself or a class of subordinate users are not covered by the above discussion. However, the protocols above are of course available to the developer(s) of a subsystem, to assure that their alterations are similarly controlled.

It is important to note that debugging of new system versions can in many cases be done compatibly on the existing system, as in a partitioned system, just as if user programs were being debugged. In certain cases involving the debugging of the lowest levels, a separate system (in space or time) must be used. However, in no case should debugging of system changes be permitted in a mode that changes the live system for all users!

9.1.2 Monitoring of Performance

Monitoring of performance is closely related to monitoring of security in terms of its implementation, but very different in its intent and information requirements.

Examples of performance monitoring include:

- real time elapsed during execution (e.g., during a command, or during a request within a command)
- chargeable units expended during execution
- measures of overhead attributable to certain classes of system functions (e.g., linkage faults, page faults)

- measures of relative execution attributable to different procedures, user-created and/or system-supported, or instructions
- measures of input-output performance
- measures of process thrashing for the system
- measures of page thrashing for the system, or for a particular subset of processes
- trace of linkage faults for a process
- trace of page faults for a process
- trace of occurrence of various classes of events.

Integrally related to performance monitoring are issues of accounting and charging for resource usage, and any restrictions on resource allocation that may ensue. Numerous examples of such restrictions involve a command being rejected or a user being interrupted due to a quota being exceeded (e.g., the maximum number of creatable processes, the maximum size of a directory, the maximum number of directories, the maximum execution time expended on a given command or operation).

As noted below in the discussions of security monitoring, such performance issues may be relevant to security, especially where denial of service is concerned. Furthermore, monitoring information must not violate security, e.g., supposedly hidden information must remain hidden.

9.1.3 Monitoring of Security Under Normal Operation

As noted above, monitoring for violations is not needed whenever the desired security is clearly covered by proved assertions for the operating system and other software, except insofar as hardware unreliability is concerned. However, monitoring of usage patterns is still desirable. For user applications, and for heavily used subsystems that are unproved, monitoring functions can be provided as a part of the program, using monitoring functions supported by the system itself. Thus monitoring may play an important initial role in a new piece of software, a role that will diminish as proofs are made.

Examples of things that might be monitored include

- excessive attempts to login as a particular user, or to login from a particular terminal
- excessive attempts to use other mechanisms requiring special authorization (e.g., changing the working directory)
- status of system parameters visible to the security officer
- occurrence of certain exception conditions
- attempts to cause denial of service (see Section 9.1.4)
- use of certain time-critical functions that must be maintained (for special users) during recovery (see Section 9.1.5)
- an audit trail of all logins, logouts, with user and terminal identification
- who is currently logged in, and what function they are executing (e.g., by symbolic procedure name)
- a trace of working directories (or attempts to change working directory) for a given process
- a trace of every procedure call for a particular process
- a log of all operator requests
- detection of resource usage patterns indicating potential misuse.

Under correct operations, certain statically proved properties will of course be satisfied. However, the detection of abnormal behavior can be facilitated by certain monitoring activities. Examples corresponding to the four principles of security are

- detection of unauthorized acquisition,
- detection of unauthorized modification,
- detection of denials of service,
- detection of loss of confinement.

In many cases this detection may be derived from more primitive testing of correct system behavior, and prevented before the security violation occurs--for example through the hardware and software redundancy of fault-tolerant design.

9.1.4 Denial of Service

In general, denial of service may be attempted in many ways, either by a single user or in collaboration with others. It may affect service to all users, or just specific users. It may cause a mild inconvenience, or may increase in seriousness to prohibit totally the use of any resources (e.g., processing, memory, input-output).

At certain places in such a multi-dimensional spectrum, denial of service becomes a security violation, and thus is of concern here. It may be made possible by a security violation not otherwise considered to be harmful (because it does not permit the unauthorized acquisition or modification of information), e.g., if a user can fill up a critical system table by creating certain objects ad nauseam. However, this example is clearly one that can be avoided by good design. In general, there should be adequate quotas on the use of each type of resource to prevent denial of service by saturation. Nevertheless, abnormal resource usage should be monitored.

It is also necessary to interrelate the usages of various types of resources, to prevent deadlocks involving multiple resources and resulting cases of denial of service (e.g., as a result of having deadlocks in critical system tables). Thus designing against deadlocks is important as are monitoring of their occurrence and recovering therefrom. Although there is much research on this problem (e.g., Dijkstra [68a], Habermann [69]), the problem is by no means solved in general. The use of a hierarchical design permits the generic proof that no deadlocks can occur between levels (again, Dijkstra [68] and Habermann [69]). However, specific proofs that no deadlocks can arise within a level are still required to guarantee avoidance of this type of denial of service, unless detection and backtracking can be guaranteed to work--which may be still more difficult. In general, a combination of good design, monitoring for deadlocks, and recovering from deadlocks seems desirable.

A somewhat simpler case (actually a special case of the above) involves making a particular vital resource unavailable (whether to one user or to all). Thus all lockouts of duration longer than expected should be monitored.

9.1.5 Security Monitoring During Recovery

In general, as soon as malfunctions are detected, it is desirable to block all user processes and to examine what the effects may have been between the occurrence of any faults and the time of the blockade. If no damage can have occurred, all processes may be restarted, assuming successful recovery. If only a subset of processes can have been affected, all others may be restarted, and further action taken with respect to the affected processes.

In certain real-time applications, it may be necessary during recovery to maintain response for certain special processes (e.g., controlling special equipment), whenever possible. In these cases, a special monitoring mode may be invoked to assure no violations of security. On the other hand, careful design may in certain cases assure that violations are so improbable as to make this monitoring mode unnecessary. For the present we are not concerned with real-time operation (although we have not excluded it), and thus we have not pursued the problem of maintaining secure response during recovery any further. Nevertheless, it can be a significant problem in certain environments.

9.2 Monitoring in the Operating System

The foregoing of course applies directly to the system described here. Monitoring exists at each level, and results in values of various existing or additional V-functions being available in certain ways. In general, the occurrence of any exception condition for any O-function or V-function may be monitored. In addition, monitoring derived V-functions may be provided to represent desired monitoring information.

In the system described here, capabilities provide the basis for specifying who has access to what monitoring information, and when. However, access to lower-level capabilities is forbidden. Specifically, it must be guaranteed that no capability can be obtained via monitoring by which the state of a lower level could be changed. Thus monitoring--whether of security or performance--can be readily shown not to compromise system security. Note that most monitoring (whether on-line or off-line) is requested at command level, although lower-level monitoring functions (V-functions) are of course invoked. The design of monitoring commands is not included here, although various commands will exist to serve the functions outlined generally in Section 9.1. Exemplary monitoring functions of lower levels that are related to security are given in Table 9.2.

Table 9.2

LOWER-LEVEL MONITORING FUNCTIONS RELATED TO SECURITY

Level 7 or above	Detection of the creation by a single process of an inordinate number of objects, e.g., segments, directories, objects, or revocable capabilities.
Level 4 and above	Detection of the occurrence of the exception condition NO_ABILITY(c, "ability"), for any desired function.
Level 3	Detection of errors in secondary storage maps (e.g., two uid's pointing to the same physical storage, or to none!)
Level 0	Detection of any reuse of unique identifiers, e.g., due to a malfunction of the system clock or uid counter, or an improper recovery. Monitoring of corrected memory errors, e.g., resulting from error-correcting codes in memory. (Note that a multiple error in a single-error correcting code can produce an apparently correct word.)

Chapter 10

SYSTEM INITIALIZATION, BACKUP, AND FAULT RECOVERY

A hierarchical system is considered to be properly initialized if each level can support all higher-level functions. In terms of the abstract machine specifications of Appendix A, the value of each V-function should be the specified initial value (in general UNDEFINED). Each level will contain initialization operations (not discussed here) that are used only during initialization. In general, each of these operations is implemented just as any other operation, using only lower-level operations. Initialization begins at level 0, proceeding upward one level at a time.

Recovery from faults is similar to initialization. In this section, we consider only hardware faults. For each valid state of an abstract machine, a set of consistent states (<u>recovery states</u>) can be described. If a fault occurs, an abstract machine can be placed in one of its recovery states by recovery procedures within the abstract machine. Part of these recovery procedures may involve calling upon lower-level abstract machines to put themselves into a recovery state. For a given abstract machine, there is an ordering among recovery states that corresponds to varying degrees of failure. For levels with large data bases (e.g., level 4), the initial state is not a satisfactory recovery state. As a result, recovery procedures at these levels seek to minimize the data loss. In our system, object backup exists solely at level 4 (in terms of segments), and is similar to the incremental techniques used in Multics. Recovery at lower levels is relatively simple.

If a level is implemented in hardware, its recovery procedures may involve reconfiguration of the resources available at that level. Relevant fault detection and reconfiguration techniques are discussed in Neumann [72] and Neumann et al. [73].

The functions for initialization, backup, and fault recovery are not included in this presentation of the design. Some of the low-level functions may in fact be implemented in hardware.

Chapter 11

CONCLUSIONS

There are many positive conclusions emerging from the work reported in the preceding chapters. There is also much future work needed to provide a definitive evaluation of this work. However, in general, the approach and the results are both very promising. Various conclusions are noted throughout the report, e.g., at the end of Chapter 1. The most notable conclusions are as follows:

- The methodology has contributed greatly to the system design, and can contribute similarly to future work--both on this system and on others. The methodology contributes to the entire spectrum of system development, including design, implementation, debugging, integration, maintenance, optimization, evolution, and management, as well as to proving properties of a system designed according to the methodology. (Early efforts on a related project at SRI for NASA-Langley show that the methodology is directly applicable to the design of an ultra-reliable airborne computing system, where properties of the fault-coverage of the system are to be proven.)

- The system design given here has enormous potential for future applications where security is crucial. In particular, the design seems well suited to the efficient solution of special security problems that have been difficult to solve on existing systems. It also has use where security is merely one of a set of system requirements (security, fault-tolerance, efficiency, flexibility, generality, ease of use, etc.). The system appears to be realistically implementable.

- Further work is needed to demonstrate the feasibility of efficient hardware implementation of the system, to verify that the system is in fact secure in the desired senses, and to evaluate the system's applicability for various security applications.

REFERENCES

	Cited
Bell and LaPadula[74] D. E. Bell and L. J. LaPadula, Secure Computer Systems: Mathematical Foundations and Model, MITRE Corp., Bedford, MA (September 1974).	1-25, C-1
Bisbey and Popek[74] R. L. Bisbey II and G. J. Popek, Encapsulation: An Approach to Operating System Security, *Proc. ACM Annual Conf.*, pp. 666-675 (December 1974).	2-1, 2-3
Boyer[74] R. S. Boyer, Private Communication, Documenting an On-Line Proof Checker (December 1974).	1-26
Branstad[73] D. K. Branstad, Privacy and Protection in Operating Systems, Report of IEEE Committee on Operating Systems Workshop, Princeton, New Jersey, Vol. 1, June 12-14, 1972. In *IEEE Computer*, Vol. 6, No. 1, pp. 43-46 (January 1973).	1-19
Bredt and Saxena[74] T. H. Bredt and A. R. Saxena, Hierarchical Design Methods for Operating Systems, *Digest IEEE Computer Society International Conference*, pp. 153-156 (September 1974).	5-20
Burke[74] E. L. Burke, Synthesis of a Software Security System, *Proc. ACM Annual Conf.*, pp. 648-50 (November 1974).	2-2, 2-4
Cosserat[72] D. C. Cosserat, A Capability Oriented Multiprocessor System for Real-Time Applications, I.C.C. Conference, Washington, D.C. (October 1972).	4-5
Cosserat[74] D. C. Cosserat, A Data Model Based on the Capability Protection Mechanism, Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, pp. 35-53 (August 1974).	4-10

Cited

Dennis and Van Horn[66] J. B. Dennis, and E. C. Van Horn, Programming Semantics for Multiprogrammed Computations, *Comm. ACM 9*, pp. 143-155 (March 1966). 1-12

Deutsch[73] L. P. Deutsch, "An Interactive Program Verifier," Ph.D. Thesis, University of California, Berkeley, California (June 1973). 3-15

Dijkstra[68a] E. W. Dijkstra, "The Structure of the THE Multiprogramming System," *Comm. ACM*, Vol. II, No. 5, pp. 341-346 (May 1968). 1-11, 9-9

Dijkstra[68b] E. W. Dijkstra, "Co-operating Sequential Processes," in *Programming Languages*, F. Genuys, ed., pp. 43-112, Academic Press (1968) 1-11

Dijkstra[68c] E. W. Dijkstra, Complexity Controlled by Hierarchical Ordering of Function and Variability, in *Report on a Conference on Software Engineering* (Randell and Naur, eds.), NATO (1968). 1-11, 3-1, 3-6

Dijkstra[72] E. W. Dijkstra, Notes on Structured Programming, in *Structured Programming* (O. J. Dahl, E. W. Dijkstra, C.A.R. Hoare), Academic Press, N.Y., pp. 1-82 (1972). 1-11, 3-1

Dijkstra[74] E. W. Dijkstra, "Guarded Commands, Non-Determinacy and a Calculus for the Derivation of Programs," Nuenen, the Netherlands (June 26, 1974). 3-14

Elspas et al.[73] B. Elspas, K. N. Levitt, and R. J. Waldinger, An Interactive System for the Verification of Computer Programs. Final Report, SRI Project 1891, Stanford Research Institute, Menlo Park, California (1973). 3-14, 3-15

Fabry[67] R. S. Fabry, A User's View of Capabilities, *ICR Quarterly Report No. 15*, ICR, U. of Chicago, Chicago, Ill., Sec. 1C (November 1967). 1-12, 4-5

Fabry[74] R. S. Fabry, Capability-Based Addressing, *Comm. ACM 17*, pp. 403-412 (July 1974). 1-12

Cited

Feiertag and Organick[71] R. J. Feiertag and E. I. Organick, The Multics Input/Output System, Proc. Third Symp. Operating Systems Principles, pp. 35-41 (October 18-20, 1971). 6-2

Floyd[67] R. W. Floyd, Assigning Meaning to Programs, Mathematical Aspects of Computer Science, Vol. 19 (J. T. Schwartz, ed.), American Mathematics Society, Providence, RI, pp. 19-32 (1967). 3-14

Gerhardt and Parnas[73] D. Gerhardt and D. L. Parnas, WINDOW: A Formally-Specified Graphics-Based Text Editor, Computer Science Department, Carnegie-Mellon, Pittsburgh, Pennsylvania (June 1973). 3-27

Glaser et al.[72] E. L. Glaser, The LOGOS System, Session of 5 Papers by E. L. Glaser, F. G. Heath, C. W. Rose, F. T. Bradshaw, S. W. Katzke, DIGEST, IEEE Computer Society Int. Conf. (COMPCON). San Francisco, pp. 175-192 (September 13, 1972). 2-5

Goldberg[74] R. P. Goldberg, A Survey of Virtual Machine Research, IEEE Computer, pp. 34-45 (June 1974). 2-3

Graham and Denning[72] G. S. Graham and P. J. Denning, Protection--Principles and Practice, Proc. AFIPS SJCC 40, pp. 417-429 (1972). 8-11

Habermann[69] A. N. Habermann,Prevention of System Deadlocks, Comm. ACM, Vol. 12, pp. 373-377,385 (July 1969). 9-9

Hoare[71] C.A.R. Hoare, Procedures and Parameters: An Axiomatic Approach, Symposium on Semantics of Algorithmic Languages. E. Engeler (Ed.) Springer Verlag (1971), pp. 102-116. 1-11

Hoare[72] C.A.R. Hoare, Proof of Correctness of Data Representations, ACTA Informatica 1, pp. 271-281 (1972). 3-9,3-28

	Cited
Hoare[74] C.A.R. Hoare, Monitors: An Operating System Structuring Concept, Comm. ACM 17, pp. 549-557 (October 1974).	1-11,A.2-2
Horning and Randell[73] J. J. Horning and B. Randell, Process Structuring, Computing Surveys, Vol. 5, No. 1, pp. 5-30 (March 1973).	5-10
Igarashi et al.[73] S. Igarashi, R. London, and D. Luckham, "Automatic Verification of Programs I: A Logical Basis and Implementation," Memo AIM-200, Stanford Artificial Intelligence Lab., Stanford, California (May 1973).	3-15
Janson[74] P. A. Janson, Removing the Dynamic Liner from the Security Kernel of a Computing Utility (Master's Thesis), MAC TR-132, MIT, Cambridge, Mass. (June 1974).	A.8-1
Jones[73] A. K. Jones, Protection Structures, Ph.D. Thesis, Carnegie-Mellon University (1973).	1-12
King[69] J. C. King, A Program Verifier, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania (September 1969).	3-15
Lampson[69a] B. W. Lampson, Dynamic Protection Structures, Proc. AFIPS 1969 FJCC, 35, AFIPS Press, Montvale, N.J., pp. 27-38 (1969).	4-5
Lampson[69b] B. W. Lampson, On Reliable and Extendable Operating Systems, Software Engineering Techniques, NATO Science Committee Working Paper (September 1969).	1-12
Lampson[73] B. W. Lampson, A Note on the Confinement Problem, Comm. ACM 16, pp. 613-614 (October 1973).	1-12,1-21 1-22,2-4 8-27
Linden[74] T. A. Linden, "Capability-Based Addressing to Support Software Engineering and System Security," Third Texas Conf. on Computing Systems, Austin, Texas, pp. 8-5-1 to 8-5-6 (November 7-8, 1974).	1-21

Cited

Lipner[74] S. B. Lipner, A Minicomputer Security Control System, COMPCON, pp. 26-28 (1974). 2-1,2-3

Manna and Pnueli[74] Z. Manna, and A. Pnueli, Axiomatic Approach to Total Correctness of Programs, Acta Informatica. 3, No. 3 (1974), 243-264. 3-14

Needham[72] R. M. Needham, Protection Systems and Protection Implications, Proc. AFIPS 1972 FJCC, 41. AFIPS Press, Montvale, N.J., pp. 571-578 (1972). 1-12,2-2 4-5

Neumann[72] P. G. Neumann, A Hierarchical Framework for Fault-Tolerant Computing Systems, Digest of IEEE Computer Society Internat. Conf. (COMPCON 72), pp. 337-340 (September 1972). 10-1

Neumann et al.[73] P. G. Neumann, J. Goldberg, K. N. Levitt and J. H. Wensley, A Study of Fault-Tolerant Computing, SRI Report (July 31, 1973). AD 766974. 10-1

Neumann et al.[74] P. G. Neumann, R. S. Fabry, K. N. Levitt, L. Robinson, and J. H. Wensley, On the Design of a Provably Secure Operating System, Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, pp. 161-175 (August 1974). A-vi

Organick[72] E. I. Organick, The Multics System: An Examination of its Structure, MIT Press, Cambridge, MA (1972). 2-2,2-3 4-5

Parnas[72a] D. L. Parnas, "A Technique for Software Module Specification with Examples," Comm. ACM 15, pp. 330-336 (May 1972). 1-5,3-3 3-8,3-29

Parnas[72b] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Comm. ACM 15, pp. 1053-58 (December 1972). 1-5,3-3 3-27

Parnas[72c] D. L. Parnas, "Some Conclusions from an Experiment in Software Engineering Techniques," Proc. FJCC, pp. 325-329 (1972). 1-5,3-27

AD-A088 601　STANFORD RESEARCH INST MENLO PARK CALIF　F/G 15/3
A PROVABLY SECURE OPERATING SYSTEM.(U)
JUN 75　P G NEUMANN, L ROBINSON, K N LEVITT　DAAB03-73-C-1454

UNCLASSIFIED　NL

3 OF 4
AD A
088601

	Cited
Parnas[72d] D. L. Parnas, "Response to Detected Errors in Well-Structured Programs," Technical Report, Department of Computer Science, Carnegie-Mellon University (July 1972).	1-5,3-9 3-27
Parnas and Siewiorek[72] D. L. Parnas, and D. P. Siewiorek, Use of the Concept of Transparency in the Design of Hierarchically Structured Systems. Technical Report, Department of Computer Science, Carnegie-Mellon University; Pittsburgh, Pennsylvania (November 1972).	1-5
Parnas[74] D. L. Parnas, "On a Buzzword: Hierarchical Structure," Information Processing 74 (IFIP), Vol. 2, pp. 336-339, North-Holland Publishing (1974).	1-11,3-27 5-11
Popek and Kline[74] G. J. Popek and C. Kline, The Design of a Verified Protection System, Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, pp. 183-196 (August 1974).	2-1,2-3
Price[73] W. R. Price, Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems, Ph.D. Thesis Carnegie-Mellon University, Department of Computer Science (June 1973).	3-7,3-27 4-5
Randell[75] B. Randell, System Structure for Software Fault Tolerance, Int. Conf. on Reliable Software, Los Angeles, California, pp. 437-449 (April 1975).	2-1
Reboh and Sacerdoti[73] R. Reboh, and E. Sacerdoti, A Preliminary QLISP Manual, Technical Note 8. Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (August 1973).	3-21
Redell[74] D. D. Redell, Naming and Protection in Extendible Operating Systems (Ph.D. Thesis UC Berkeley), MAC TR-140, MIT, Cambridge, Mass. (November 1974).	A.4-3
Redell and Fabry[74] D. D. Redell and R. S. Fabry, Selective Revocation of Capabilities, Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, France, pp. 197-209 (August 1974).	1-13,4-11 8-19

Cited

Robinson[73a] L. Robinson, Design and Implementation of a Multilevel System Using Software Modules. Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania (July 1973). 3-27

Robinson[73b] L. Robinson, Hierarchical Proof of TREESORT. SRI unpublished paper (November 1973). 3-24,3-28

Robinson and Holt[73] L. Robinson and R. C. Holt, Formal Specifications for Solutions to Synchronization Problems. SRI Report, Computer Science Group (November 1973). 3-27

Robinson and Levitt[75] L. Robinson and K. N. Levitt, Proof Techniques for Hierarchically Structured Programs, SRI (January 1975). Submitted for publication. 3-1,3-28 4-10

Robinson et al.[75] L. Robinson, K. N. Levitt, Peter G. Neumann, A. R. Saxena, On Attaining Reliable Software for a Secure Operating System, Int. Conf. on Reliable Software, pp. 267-284, Los Angeles, California (21-23 April 1975). 1-3,2-6

Rulifson et al.[72] J. F. Rulifson, J. A. Derksen, and R. J. Waldinger, QA4: A Procedural Calculus for Intuitive Reasoning. Technical Note 73, Artificial Intelligence Center, Stanford Research Institute, Menlo Park, California (1972). 3-21

Saltzer[74] J. H. Saltzer, Ongoing Research and Development on Information Protection, ACM Operating Systems Review 8, pp. 8-24 (July 1974). 2-2,2-4

Saxena and Bredt[75] A. R. Saxena and T. H. Bredt, A Structural Specification of a Hierarchical Operating System, Proc. 1975 International Conference on Reliable Software, pp. 310-318 (April 1975). 5-18

Schroeder[72] M. D. Schroeder, Cooperation of Mutually Suspicious Subsystems in a Computer Utility, Ph.D. Thesis, MIT (September 1972). MAC TR-104. 1-20

Cited

Sevcik et al.[72] K. C. Sevcik et al., Project SUE as a Learning Experience, Proc. AFIPS 1972 FJCC, 41, AFIPS Press, Montvale, N.J., pp. 331-339 (1972). 1-12

Simon[62] H. A. Simon, "The Architecture of Complexity," Proc. Am. Phil. Soc., Vol. 106, pp. 467-82 (December 1962). 1-11

Spitzen[74] J. M. Spitzen, Approaches to Automatic Programming (Ph.D. Thesis), Center for Research in Computing Technology, Harvard University, Cambridge, Mass. (May 1974). 3-28

Sturgis[73] H. E. Sturgis, A Postmortem of a Time-Sharing System, Ph.D. Thesis, University of California, Berkeley (1973). 1-12

Waldinger and Levitt[73] R. J. Waldinger, Reasoning about Programs, Proc. SIGACT/SIGPLAN Symposium on Principles of Programming Languages, October 1-3, 1973. Boston, Massachusetts (1973); also in Artificial Intelligence, Vol. 5, pp. 235-316 (1974). 3-21

Wegbreit and Spitzen[75] B. Wegbreit and J. M. Spitzen, Proving Properties of Complex Data Structures, Submitted for publication (1975). 3-27

Weissman[69] C. Weissman, Security Controls in the ADEPT-50 Time Sharing System, FJCC 1969, pp. 119-133. C-1

Wulf et al.[74] W. A. Wulf et al., HYDRA: The Kernel of a Multiprocessor Operating System, Comm. ACM 17, pp. 337-345 (July 1974). 1-12,2-2 2-4,4-3 4-5,A.5-1

Appendix A

SPECIFICATIONS FOR THE SYSTEM

The structure of the system is outlined in Table 1.1, and discussed in Chapter 5. The functions visible to the user-interface to the operating system (level 10) are summarized in Table 6.1. This appendix provides descriptions and formal specifications for all functions visible at the interface to each level (up through level 10), along with certain non-visible (hidden) V-functions whose existence simplifies the specifications of the visible functions. There is one section for each level, including a written description of the functions at that level and specifications for that level. The levels are given in order of increasing level number, Sections A.0 to A.10 corresponding to levels 0 to 10, respectively. The functions specified here are summarized in Tables A.0 to A.10 for levels 0 to 10, respectively. It is recommended that the descriptions preceding the specifications be read bottom-up, i.e., from A.0 to A.10. The reader may then wish to read the specifications in a different order, e.g., top-down (A.10 to A.0). In that lower levels are more illustrative of possible implementations than higher levels, some readers may wish to begin at level 4 and work upward to level 10. In general, those functions forming the user-visible operating system interface (i.e., Table 6.1) are the most important functions to be understood.

Specifications for functions relating to establishing special security environments, to monitoring of security and performance, and to initialization, recovery and fault-tolerance are not included in this report.

The language and notation used for the specifications are given next. (For further background, see Chapter 3.)

The module specifications of Sections A.0 to A.10 are written in terms of assertions. The assertion language closely resembles predicate calculus, so that only its departures from predicate calculus are explained. The primitive types are integers (i), booleans (b), capabilities (c), unique identifiers (u), and machine words (w). These abbreviations are used in lieu of declarations. V-functions may return the distinguished value UNDEFINED for some subset of their domain. The compound structures of sets and tuples also exist. The following primitives apply to sets (where a is an element and S is a set).

$b = a \in S$ (elementhood)

$i =$ cardinality (S) (number of elements in S)

$\{a|P(a)\}$ = [the set of all "a" having property P(a)]

$\{a\}$ = (a set of arbitrary elements each of which is of the same type as "a").

The following primitives apply to tuples (where a is an element, S is a set, and T is a tuple):

$b = a \in T$ (elementhood)

i = length(T) (length of tuple T)

T = set_to_tuple(S) (a tuple made from set S)

$a = T[i]$, $1 \le i \le \text{length}(T)$ (indexing)

[a] = (a tuple of arbitrary elements each of which is of the same type as "a").

The scope of a variable is limited to the function in which the variable is declared: A variable is declared by either:

(1) being a formal parameter to the function, or

(2) following any of the symbols $\forall$, $\exists$, LET, or CHOOSE.

We assume the expressions

$\forall x(Px)|[Q(x)]$ and

$\forall x(P(x) \supset Q(x))$

to be equivalent, except that in the former expression, the predicate P(x) must be defined over all x, but the predicate Q(x) need only be defined for values for which P(x) is true. "LET x = exp" simply defines a new variable "x" to be replaceable by the expression "exp," in the fashion of a macro. "CHOOSE x (assertion (x))" selects exactly one value of x satisfying the properties of "assertion(x)". This is an imperative statement for use in the "EFFECTS" sections of O-functions.

The specifications of a level contain four parts: PARAMETERS, DEFINITIONS, EXCEPTIONS, and FUNCTIONS. The PARAMETERS section contains type descriptions for names of variables that appear later in the specifications. Also appearing in the PARAMETERS section are some module values that depend on the particular instance of the module being implemented (e.g., maximum segment size). The DEFINITIONS section contains macros global to all of the functions of the level. The EXCEPTIONS section contains the macro definitions of exception conditions for the O- and V-functions of that level, saving the writer the ordeal of repeating exception conditions that appear in many functions. All the V-function values in the EXCEPTIONS apply to values before the call. Names of exception conditions are given in upper case.

The FUNCTIONS section of a module specification contains the assertions that formally specify each function of the system. Specifications of HIDDEN V-functions have the following parts: PURPOSE and INITIAL VALUE (denoted as "INITIALLY"). PURPOSE states the significance of the function in natural language. INITIALLY is an expression characterizing the initial value for the function. Non-HIDDEN V-functions have an EXCEPTIONS section as well. The EXCEPTIONS are assumed to be tested in order; the first one flagged corresponds to the exception routine invoked. Some non-HIDDEN V-functions are DERIVED; and these functions have a DERIVATION section, an expression to state how they can be derived from other (non-DERIVED) V-functions. The specifications of O- and OV-functions have many of the same parts as those of V-functions. However, the EFFECTS section is unique to O- and OV-functions. The EFFECTS are assertions containing V-function values before the call (in single quotes: '...') and V-function values following the return (unquoted). The EFFECTS of an O-function call are all assumed to occur simultaneously. All V-function values not explicitly referenced in the EFFECTS section remain the same for an O-function call.

In certain low-level O-functions (namely at levels 0 and 2), a DELAY section is included specifying that the effects cannot take place UNTIL the condition given in the DELAY section is satisfied. DELAY implies that some other process must first act to make the UNTIL condition TRUE. (It is of course desirable to prove at the given level that the stated condition must eventually occur.)

Primitive functions of the specification language are also designated as upper-case letters and symbols, e.g., IF, THEN, ELSE, TRUE, FALSE, LET, CHOOSE. An asterisk on the right-hand side of an effect section (in an O-function) is used as an abbreviation for the left-hand-side function in quotes. If no arguments are given, the arguments are those of the left-hand side.

Examples:

```
                function(x,y,z) = * + w;
```

is equivalent to writing

```
                function(x,y,z) = 'function'(x,y,z) + w;
```

while

```
                function(x,y,z) = *(x,y,z+w);
```

is equivalent to writing

```
                function(x,y,z) = 'function'(x,y,z+w).
```

Certain levels are defined in terms of state representations, whereby several (hidden) V-functions are collected into a single tuple-valued function. Although this may put an initial burden on the reader to remember or look back for the order of the functions in the tuple, it greatly simplifies the specifications of O-functions. The asterisk convention also applies componentwise. For example, a triple-valued V-function "[f1,f2,f3] = function(x,y)" may appear in an O-function effect as

```
          function(x,y) = [*,*+3,*],
```

equivalent to

```
          function(x,y) = ['f1'(x,y), 'f2'(x,y) + 3, 'f3'(x,y)]
```

i.e.,

```
          f1(x,y) = *
          f2(x,y) = *+3
          f3(x,y) = * .
```

Some of the arguments to functions are not explicitly provided by the caller, but are implicitly provided by the system. Implicit arguments "a" are denoted by angle brackets as "⟨a⟩".

As of the present writing, we are still experimenting with various different styles for specifications. Thus, for example, level 6 uses a state representation for the set of hidden V-functions associated with each entry, and treats it as a single tuple-valued hidden V-function. For other levels, each hidden V-function is specified separately. We have found many tradeoffs in the various ways of writing specifications. For example, the style of level 6 seems easiest to write, somewhat harder to read the first time, but then much easier to read once understood.

An important aspect of this approach is the ease with which the design can be (and has been) revised. Because each level is specified independently from other levels, changes in level dependency are easily handled. Furthermore, most of the design improvements made over the last few months (typically involving a single level) have been reflected in changes of only a few lines in the specifications of one or two functions. As of this writing, there may still be inconsistencies in the specifications, although most of these appear to be notational or stylistic. (For historical comparability with an earlier preliminary design, see Neumann et al. [74].)

A.0 Level 0: Capabilities, Addressing and Interrupts

Level 0 is the most primitive level specified here. Together with level 1, it is presumably implemented entirely in hardware. Level 0 handles capability creation, detection of interrupts, primitive address mapping, and the most primitive functions used by other levels (e.g., arithmetic instructions). Indexing and indirection are handled at level 1 through which level all instructions are executed. When a capability is created, all access bits are set to 1. The derived V-function "restrict_access(c,i)" may be used to change any access bit(s) of the capability c to 0. When an interrupt is recognized, the machine gets into an interrupt mode and transfers control to a predetermined location. The location is determined by the kind of interrupt detected. The interrupt signal is to be sent by an asynchronous device external to the processor. The routine beginning at the location to which the control is transferred is expected to save the current state and transform the interrupt into a signal operation on a condition variable. This routine operates at level 2. The interrupt system is disabled whenever an interrupt signal is recognized by level 0, and has to be enabled by the interrupt routine after the state is saved. The interrupt system is enabled in the "normal" mode and disabled in the "interrupt" mode.

At this level, interrupts may be masked and unmasked under program control. The masks on the interrupts are part of the state of a process. An interrupt which is masked is recorded, but is not recognized until it is unmasked.

Primitive Address Mapping

The basic address format is referred to as an offset capability f. It is applicable only for segments. It is a structure which consists of the following fields: abilities, uid, page no, displacement, and indirection.

The syntax for referring to a particular field of an offset capability f is "field(f)", e.g., abilities(f) means the "abilities" field of the offset capability f. The page_no field concatenated with the displacement field is referred to as the offset field. It is permissible to perform add and subtract operations on the offset field but not on the access or uid fields. From the offset, the page_no can be extracted by the function "page_no(offset)". The search of the cap_map_table, and the adr_map_table (described below) is performed using the uid and the page_no of an offset capability.

As an abbreviation, we will write f + i to mean that integer i is added to the offset part of f.

Level 0 contains two mappings for calculating physical addresses from segment unique id's and page number. A virgin uid is one that refers directly to a segment. The first mapping is a capability map, which maps from revocable id's to virgin uid's (and from virgin id's onto themselves--for simplicity of specification, although not necessarily in implementation). The second memory mapping is an address map from virgin id's and page numbers to start addresses for the page. This feature allows the handling of revocation without removing the page entry from the table (and the page from core). The tables can be implemented by an associative memory, by core tables, or by a combination of both.

Address Translation

The address translation takes place at level 0, in hardware. It uses the cap_map and adr_map mentioned earlier. The uid of the offset capability is matched against the entries in the cap_map. If no match is found, then the cap_map fault is signaled back. If a match is found, then the associated entry in the table is the root_id of the object. The adr_map is searched for a match against the root_id concatentated with page_no. If no match is found, then an address_map fault is returned. If a match is found, then the associated entry is the starting address of the page in the main memory. To this address the displacement is added to obtain the required main memory address.

To insert a page address into the address map, the function enter_adr_map(u,h,adr,bounds) is used. This associates the start address adr with the page identification pair (u,h) and the address bound for the page (of interest primarily for the last page of the segment). Here u is the virgin segment unique id, and h is the page number within segment u. Boundary checks for addressing beyond the address bound can be made against bounds(u,h). When the page is removed, delete_adr_map(u,h) is called. The unique id of the segment first making a page reference is inserted into the capability map by calling enter_cap_map(uu,vu). The usage uid uu is associated with the virgin uid vu. When revocation takes place all usage uid's for virgin uid u are deleted by calling delete_cap_map(u). All usage uid's must be deleted because level 4 doesn't have back pointers to determine which uid's should be revoked. Thus, all uid's are temporarily invalidated and relinking of the valid uid's through level 4 can take place when access is desired.

Address translation takes place when the hardware chains through both tables to get the desired address. Several errors can occur in address translation: BOUNDS_CHECK, an out-of-bounds address; CAP_MAP_FAULT, a missing entity in the capability map; and ADDRESS_MAP_FAULT, a missing entry in the address map.

All storage is treated uniformly in this fashion, even if invisible to higher levels, e.g., as in the case of the level 2 storage which is invisible to levels 3 and above. Storage for levels below 3 is maintained in a partition of the maps that, once initialized, is not changed. (This is not shown in the specifications.)

The functions of level 0 and their specifications are given in Table A.0.

Table A.0

FUNCTIONS OF LEVEL 0

Purpose	V-Functions	OV- and O-Functions

Capabilities	{c} = h_cap_set	c = create_cap
		cl = restrict_access(c,i)

Interrupts	{u} = h_interrupt_set^{0}
	b = mask(u)^{0}
	b = interrupt(u)^{0}
	b = mode^{0}
		set_mask(c)^{2}
		reset_mask(c)^{2}
		set_interrupt(c)^{2}
		reset_interrupt(c)^{2}
		set_mode_normal(c)^{2}

Address Translation	ul = entry_cap_map(u)^{1}
	adr = entry_adr_map(u,h)^{1}
	i = bounds(u,h)^{1}
	b = h_entry_adr(adr)^{0}
	b = h_entry_cap(u)^{0}
	i = available_cap_map_space^{1}
	{u} = cap_map^{1} /D/
	{root_id} = adr_map^{1} /D/
	i = h_read(adr)^{0}
		enter_cap_map(uu,vu)^{1}
		delete_cap_map(u)^{1}
		enter_adr_map(u,h,adr,i)^{1}
		delete_adr_map(u,h)^{1}
		set_bounds(u,h,i)^{1}
		write(f1,f2)^{1}
		/no indirection/

Note: A superscript denotes the maximum level at which the function is to be accessible, if such a restriction exists. "/D/" denotes derived V-functions. "0" denotes a hidden function.

PARAMETERS FOR LEVEL 0

b: boolean

c: capability = (abilities(c)||uid(c))

u: unique_id

h: page_no

f: offset capability c||offset||b /b is ignored at level 0/
/offset = page_no||displacement/

i: integer

adr: address in main memory
/0 ≤ adr ≤ max_adr/

uu: uid /usage uid/

vu: uid /virgin uid/

DEFINITIONS FOR LEVEL 0

root_id(f) = [entry_cap_map(uid(f))||page_no(f)]

adr(f) = [entry_adr_map(root_id(f))||displacement(f)]

EXCEPTION CONDITIONS FOR LEVEL 0

INVALID_MASK({c}): ∃ cl ∊ {c} ∋ uid(c) ∉ interrupt_set

INVALID_MASK_ABILITY{c},"a"): ∃ cl ∊ {c} ∋ NO_ABILITIES(cl,"a");

MASKED({c}): ∃ cl ∊ {c} ∋ 'mask'(uid(cl)) = TRUE;

UNMASKED({c}): ∃ cl ∊ {c} ∋ 'mask'(uid(cl)) = FALSE;

UNSETTABLE(u): u ∊ 'interrupt_set' ∨ 'interrupt'(u) = FALSE;

UNRESETTABLE(u): u ∊ 'interrupt_set' ∨ 'interrupt'(u) = TRUE;

INVALID_MODE: 'mode' = FALSE;

ENTRY_EXISTS(uu): 'entry_cap_map'(uu) = DEFINED;

UNAVAILABLE: 'available ≤ 0;

ADR_EXISTS(adr): 'h_entry_adr'(adr) = TRUE;

UNDEFINED_UID(u): 'h_entry_cap'(u) = UNDEFINED;

INVALID_ADR(adr): ¬ (0 ≤ adr ≤ max_adr);

DEFINED_ENTRY(u,h): 'entry_adr'(u,h) = DEFINED;

UNDEFINED_ENTRY(u,h): 'entry_adr_map'(u,h) = UNDEFINED;

INVALID_DISPLACEMENT(i): ¬ (0 ≤ i ≤ max_page_length);

CAP_ERROR(f): uid(f) ∉ 'cap_map';

ROOT_ID_ERROR(i): root_id(i) ∉ 'adr_map';

ADR_ERROR(f,i): displacement(f) > bounds(root_id(i));

NO_ABILITIES(c,"ac"): "ac" ∉ 'abilities'(c);

HIDDEN V-FUNCTION {c} = h_cap_set

PURPOSE: What is the set of all capabilities that have ever existed?

INITIALLY: {unit_cap_set}

V-FUNCTION: {u} = h_interrupt set

PURPOSE: What is the set of possible interrupts ?

INITIALLY: {init_i}

V-FUNCTION: b = mask(u)

PURPOSE: Is the interrupt u is masked?

INITIALLY: IF u ∈ {init_i} THEN FALSE ELSE UNDEFINED

VISIBLE V-FUNCTION: b = interrupt(u)

PURPOSE: Is an interrupt to be signaled? The interrupt is signaled on transition from false to true.

INITIALLY: IF u ∈ {init_i} THEN FALSE ELSE UNDEFINED

VISIBLE V-FUNCTION: b = mode

PURPOSE: Is the mode "interrupt"? /TRUE = "interrupt", FALSE = "normal"/

INITIALLY: FALSE

VISIBLE V-FUNCTION: u1 = entry_cap_map(u)

PURPOSE: value of the cap_map_entry for the uid u

INITIALLY: ∀u[UNDEFINED]

EXCEPTIONS: NONE

VISIBLE V-FUNCTION: adr = entry_adr_map(u,h)

PURPOSE: value of the adr_map_entry for the page h of the segment u segment_id, page_id in referred to as the key.

INITIALLY: ∀(u,h)[UNDEFINED]

VISIBLE V-FUNCTION: i = bounds(u,h)

PURPOSE: maximum valid address in the page h of the segment u

INITIALLY: $\forall$(u,h)[UNDEFINED]

HIDDEN V-FUNCTION: b = h_entry_adr(adr)

PURPOSE: Does the main_memory_address, adr, have an entry in the adr_map?

INITIALLY: $\forall$adr[FALSE]

HIDDEN V-FUNCTION: b = h_entry_cap(u)

PURPOSE: Is there an entry for the virgin uid u in the cap_map?

INITIALLY: $\forall$u[FALSE]

V-FUNCTION: i = available

PURPOSE: How many spaces are available in the cap_map?

INITIALLY: to be defined later (system dependent)

DERIVED V-FUNCTION: {u} = cap_map

PURPOSE: What is the set of uid's for which entries exist in the cap_map_table?

DERIVATION: {u $\ni$ entry_cap_map(u) = DEFINED}

DERIVED V-FUNCTION: {root_id} = adr_map

PURPOSE: What is the set of root_ids for which entries exist in the adr_map_table?

DERIVATION: {u $\ni$ entry_adr_map(u) = DEFINED}

HIDDEN V-FUNCTION: i = h_read(adr)

PURPOSE: What is the value in the memory location identified by adr?

INITIALLY: $\forall$adr[UNDEFINED]

OV-FUNCTION c = create_cap

PURPOSE: create a new capability with a new uid, and with abilities ALL.

```
EFFECT:    CHOOSE c, u ∋ [c ∉ 'h_cap_set';
             uid(c) = u;
             abilities (c) = ALL;
             h_cap_set = 'h_cap_set' + c]
```

```
OV-FUNCTION  cl = restrict_access(c,i)

PURPOSE:  create a capability with restricted abilities (but with same uid as
          given).     /Note abilities cannot be increaased./

EFFECT:   CHOOSE cl ∋ [cl ∉ 'h_cap_set';
                     uid(cl = uid(c);
                     abilities(cl) = abilities(c) ∧ i;
                     h_cap_set = 'h_cap_set' + cl]
              /Note duplicate capabilities may thus be created, but are
               considered as distinct./


O-FUNCTION:  set_mask({c})

PURPOSE:  to disable interrupts {uid(c)}

EXCEPTIONS:  INVALID_MASK({c});
             INVALID_MASK_ABILITY({c},"set_mask");
             MASKED({c});

EFFECT:  ∀cl ∊ {c}[mask(uid(c)) = TRUE];


O-FUNCTION:  reset_mask({c})

PURPOSE:  to enable the interrupts {uid(c)}

EXCEPTIONS:  INVALID_MASK({c});
             INVALID_MASK_ABILITY({c},"reset_mask");
             UNMASKED({c});

EFFECT:  ∀cl ∊ {c}[mask(uid(cl)) = FALSE]


O-FUNCTION:  set_interrupt(c)

PURPOSE:  to signal an interrupt and transfers control to a fixed location

LET u = uid(c)

EXCEPTIONS:  UNSETTABLE(u)
             NO_ABILITY(c,"set_interrupt")

DELAY:  [mask(u) = TRUE] OR ['mode' = TRUE]

EFFECT:   mode =   TRUE;
            pc_interrupt = u;
            interrupt(u) = TRUE;
```

O-FUNCTION: reset_interrupt(c)

PURPOSE: to enable the interrupt to be set

LET u = uid(c)

EXCEPTIONS: UNRESETTABLE(u);
NO_ABILITY(c,"reset_interrupt");

EFFECT: interrupt(u) = false

O-FUNCTION: set_mode_normal(c)

PURPOSE: to enable interrupts, i.e., change from interrupt mode to normal mode

EXCEPTIONS: NO_ABILITY(c,"set_mode");
INVALID_MODE;

EFFECT: mode = FALSE;

O-FUNCTION: enter_cap_map(uu,vu)

PURPOSE: to add an entry to the cap_map. The entry will map the usage uid uu onto the virgin_id(root_id)=vu. An attempt to add an entry for an uid uu, that already has an entry will cause an error. An error indication will also be returned for map overflow.

EXCEPTIONS: ENTRY_EXISTS(uu);
UNAVAILABLE;

EFFECT: available = 'available' - 1;
entry_cap_map(uu) = vu;
h_entry_cap(vu) = TRUE;

O-FUNCTION: delete_cap_map(u)

PURPOSE: to delete all entries from the cap_map, which are mapped onto u. An attempt to delete undefined entries will cause an error.

EXCEPTION: UNDEFINED_UID(u);

EFFECT: ∀uu('entry_cap_map(uu)' = u)
[entry_cap_map(uu) = UNDEFINED;
available = 'available' + cardinality{uu}];
h_entry_cap(u) = FALSE;

O-FUNCTION: enter_adr_map(u,h,adr,i)

PURPOSE: to add an entry to the adr_map. The entry will map the page number h of the segment u onto the main_memory address, adr. An attempt to add an entry for a page that already has an entry will cause an error, as will an attempt to assign the same adr to two different pages. The bounds for the page will be set to i.

EXCEPTIONS: INVALID_ADR(adr);
DEFINED_ENTRY(u,h);
ADR_EXISTS(adr).
INVALID_DISPLACEMENT(i);

EFFECTS: entry_adr_map (u,h) = adr;
h_entry_adr(adr) = TRUE;
bounds(u,h) = i;

O-FUNCTION: delete_adr_map(u,h)

PURPOSE: to delete an entry from adr_map

EXCEPTIONS: UNDEFINED_ENTRY(u,h);

EFFECTS: entry_adr_map(u,h) = adr
h_entry_adr(adr) = FALSE;
bounds(u,h) = UNDEFINED;

O-FUNCTION: set_bounds(u,h,i)

PURPOSE: to set the maximum page displacement for the page h of the segment u

EXCEPTION: UNDEFINED_ENTRY(u,h);
INVALID_DISPLACEMENT(i);

EFFECT: bounds((u,h)) = i

O-FUNCTION: write(f1,f2)

PURPOSE: to write the contents of the memory address mapped onto by uid(f1), into the memory address mapped onto by uid(f2).

EXCEPTIONS: CAP_ERROR(f1);
CAP_ERROR(f2);
ROOT_ID_ERROR(1);
ROOT_ID_ERROR(2);
ADR_ERROR(f1,1);
ADR_ERROR(f2,2);
NO_ABILITIES(c1,"read");
NO_ABILITIES(c2,"write");

EFFECT: h_read(adr(f2)) = 'h_read'(adr(f1))

A.1 Level 1: Generalized Memory Addressing

Level 1 is an extension of the hardware and handles machine instructions. Most of the machine instructions will be mapped directly on to the level below (level 0). The machine instructions that are of special concern here are indirect load/store, move, push, pop, call, return, write-out-of, and write-into. These instructions require multiple accesses to main memory and have to be at a higher level than level 0, so that they can be interrupted between successive accesses to the main memory. (Level 0 contains interrupts and single accesses to main memory as indivisible operations.) The V-functions and O-functions for address translation (see Table A.0) at level 0 are replicated at level 1. We now describe the effective address calculation, using the address translation in level 0.

Effect Address Calculation

The format for the generalized address of a segment is a = (f || x),

> where f is an offset capability, f = (c || offset || b), where "c" is a capability, "offset" in the address within the segment, and "b" is the indirection bit; x is a capability for an index register.

The contents of the index register are added to the offset part of f before indirection, indirection taking place if indirection(f) = 1. Indirection gives a new generalized capability, for which the effective address calculation process is repeated until no further indirection remains (or the permissible maximum depth is reached). At this point the ultimate effective address e(a) is obtained.

At this level we consider registers to be memory locations. All operations consist of transferring the contents of one memory location to another. Usually one of the memory locations referred to will be a register. At this level we do not have to specify the number of registers in our machine. That is decided at implementation time.

We allow many levels of indirection, up to a maximum number of levels, indirect_max, which is fixed in implementation. A limit on the number of levels of indirection is necessary to prevent indefinite looping, and to guarantee that all operations will terminate in finite time. The function "write (a1,a2)" allows the contents of the memory location denoted by the effective address of a2 to be the same as the contents of the memory location denoted by the effective address of a1.

The function "move(a1,a2,n)" allows the contents of n locations beginning at a2 to be changed to the contents of n locations beginning at a1 in a single instruction. This is useful in moving contiguous data from one place to another.

We assume the existence of two hardware-interpreted stacks, which are referred to implicitly. One of them is used for passing parameters (and is called the parameter stack) to procedures and the other (called return stack) keeps track of the return addresses and other information inaccessible to a procedure (i.e., a called procedure can only read or write into the parameter stack). It can refer to the return stack only via a return instruction.

The two stacks maintain a data structure for the current procedure activation called a frame. A frame is a subsegment of the parameter stack accessible to the current activation (all other stack locations are inaccessible). It is delimited by an upper bound and a lower bound. A frame is used by a procedure activation to receive parameters from its calling activation, to house local storage, and to pass parameters to procedures called by the activation. Local maintenance of the frame during the activation (besides write-out-of and write-into) is accomplished by the instructions "push" and "pop"; "push(a,n)" places n entries (from starting address a) on the stack above the upper bound. The upper bound is increased by n (see Figure A.1a). In the illustrations, UB stands for "upper bound," and LB for "lower bound." The current frame of the stack is shaded, and old values are in single quotes. The operation "pop(a,n)" removes n elements from the stack and places them in contiguous locations starting at a. The upper bound is decremented by n (Figure A.1b).

In order to call a procedure, a program puts the parameters on the stack using the operation "push". The parameters are now sitting at the top of the current frame. The instruction "call(a,n)" transfers control to the location specified by the capability e(a) (with offset = 0), and changes the lower bound of the stack to the upper bound decremented by n. This enables the new activation to have a frame consisting of n entries (see Figure A.1c). "return(n)" changes the lower bound to the lower bound of the previous frame and changes the upper bound to be n places above the current lower bound. This means that the bottom n places of the called environment are returned to the calling environment (Figure A.1d).

The capability for the parameter stack is stored in the register "p_stack"; the registers "p_stack_top" and "p_stack_bottom" contain pointers to the top (upper bound) and the bottom (lower bound) of the parameter stack, respectively.

The capability for the return stack is stored in the register "r_stack", the pointer to the top of the stack is in the register "r_stack_top".

The functions of level 1 and their specifications are given in Table A.1.

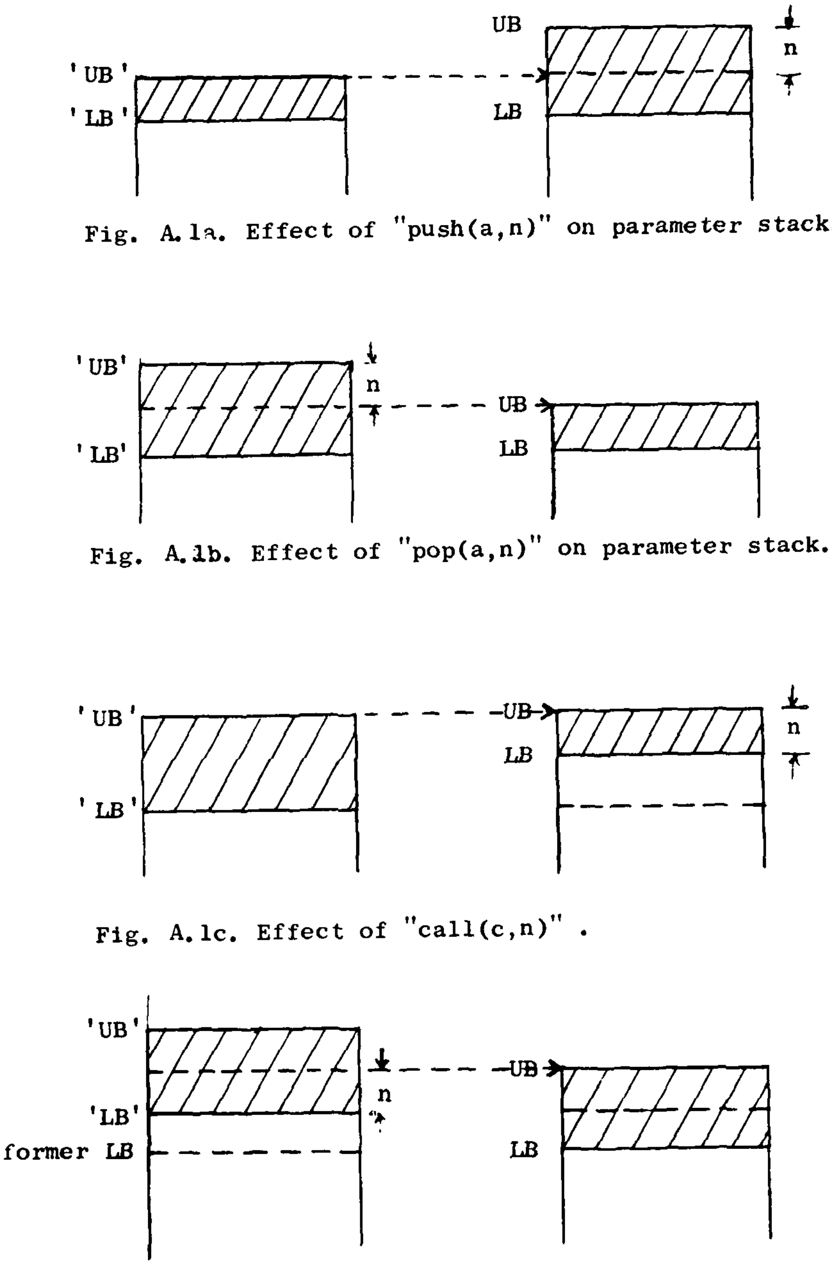


Fig. A.1a. Effect of "push(a,n)" on parameter stack.

Fig. A.1b. Effect of "pop(a,n)" on parameter stack.

Fig. A.1c. Effect of "call(c,n)" .

Fig. A.1d. Effect of "return(n)"

FIGURE A.1 EFFECTS OF "push", "pop", "call", AND "return"

Table A.1

FUNCTIONS OF LEVEL 1

V-functions	O-functions
j = h_read(adr)^4	write(a1,a2)^4
j = f_read(f)^4	move(a1,a2,n)^4
(j1 || j2) = a_read(f)^4	push(a,n)^4
	pop(a,n)^4
	call(c,n)^2
	return(n)^2
	write_out_of(a,i)^4
	write_into(a,i)^4

Note 4" restricted to levels 4 and below.

<u>Parameters for Level 1</u>

j: contents of a memory location

adr: machine address of a memory location

c: capability

f: offset-capability

f = (capability (f) || offset (f) || indirection (f)), where
capability (f) = (abilities (f) || uid (f)) and
offset (f) = (page_no (f) || displacement (f))

indirect (f): boolean

offset (f): integer /0 $\leq$ offset (f) $\leq$ maximum segment size/

x: index

a: generalized address a = (f || x) = f(a) || x(a)

n: integer /stack increment or decrement/

i: integer

Level 1 Definitions

```
f + i = (capability(f) || offset(f) + i);

indirect(a) = [indirection(f(a) = 1];

indexed_address(a) = f(a) + 'f_read'(x(a));

a + i = indexed_address(a) + i

root_id(f) = [entry_cap_map(uid(f)) || page_no(f)];

adr(f) = [entry_adr_map(root_id (f)) || displacement(f)];

u(a,m) = ∀m(0 ≤ m ≤ indirect_max)
     IF m = 0 THEN       indexed_address(a)
     ELSE [IF indirect(u(a,m-1) THEN
          indexed_address(a_read(u(a,m-1)
          ELSE u(a,m)];

e(a) = u(a, indirect_depth(a)); /effective address/
indirect_depth(a) = min i ∋ (u(a,i) = u(a, indirect_max));

/indirect_max = integer established in hardware/;

gp_stack = 'f_read'(p_stack) || 'f_read'(p_stack_top);

gr_stack = 'f_read'(r_stack) || 'f_read'(r_stack_top);

gp_frame_length = 'f_read'(p_stack_top) - 'f_read'(p_stack_bottom );
```

Level 1: Exception Conditions

```
cap_error(f): uid(f) ∉ cap_map;

adr_error(f): root_id(f) ∉ adr_map;

offset_error(f): displacement(f) > bounds(root_id(f));

overflow(f): offset(f) > maximum_segment_size

no_ability(f,"a"): "a" ∉ abilities(f);

indirect_error(u(f,indirect_depth(f)):

     ¬ ∀m(0 < m ≤ indirect_depth(f))
     [uid(u(f,m)) ∈ cap_map
     ∧ root_id(u(f,m)) ∈ adr_map
     ∧ displacement(u(f,m)) ≤ bounds(entry_adr_map(root_id(u(f,m)))]

indirect_read_error(u(f,k):

     ¬ ∀m(0 < m ≤ k) no_abilities(u(f,m), "read");

nonpositive(n): n ≤ 0;

move_error(f,n):

     ¬ ∀m(0 ≤ m < n)
     [uid(f) ∈ cap_map
     ∧ root_id(f+m) ∈ adr_map
     ∧ displacement(f+m) ≤ bounds(entry_adr_map(root_id(f+m))];

stack_underflow(p_stack):

     'f_read'(p_stack_top) - n < 'f_read'(p_stack_bottom);

invalid_index(i):  ¬ [0 < i < gp_frame_length];

indirection_error(a):  indirect_depth (a) = indirect_max;

INVALID_OFFSET(f):  offset(f) ≠ 0;
```

(Note: all names of exception conditions should be upper case throughout.)

EXCEPTION MACROS:

```
write_exceptions_1(a):

    [cap_error(a);
     adr_error(a);
     offset_error(a);
     overflow(a);
     indirect_error(e(a)):
     indirect_read_error(e(a));
     indirect_error(a)]


write_exceptions_2(a):
    [cap_error(a);
     adr_error(a);
     offset_error(a);
     overflow(a);
     indirect_error(e(a));
     indirect_read_error(e(a)_1);
     indirection_error(a);
     no_abilities(e(a),"write")]
```

Level 1: Specifications

V-function: j=h_read(adr)

Purpose: What is the value in the given main memory location:

Initial Value: ∀ adr undefined

Exceptions: None

Derived V-function: j=f_read(f)

```
Exceptions:  cap_error(f);
             adr_error(f);
```

Value: h_read(adr(f));

Derived V-function: j1 || j2 = a_read(f)

```
Exception:  cap_error(f)
            adr_error(f)
            adr_error(f+1)
```

Value: h_read(adr(f)) || h_read(adr(f+1))

O-function: write(a1,a2)

Purpose: To write the contents of a1 into a2.
The contents of a1 remain unchanged.

Exceptions:

```
write_exceptions_1(a1);
write_exceptions_2(a2);
```

Effect:

```
f_read(e(a2)) = 'f_read'(e(a1));
```

O-function: move(a1,a2,n)

Purpose: To write the contents of n locations beginning at a1 into n locations beginning at a2.

Exceptions:

```
write_exceptions_1(a1);
write_exceptions_2(a2);
overflow(e(a1) + n-1);
overflow(e(a2) + n-1);
move_error (e(a1),n);
move_error(e(a1),n);
```

Effect

```
∀m(0 ≤ m < n)f_read(e(a2) + m) = 'f_read'(e(a1) + m);
```

O-function: push(a,n)

Purpose: to push n elements from location a onto the parameter stack.

Exceptions:

```
write_exceptions_1(a);
nonpositive(n);
no_ability(gp_stack,"write");
overflow(gp_stack + n);
move_error(e(a),n);
move_error(gp_stack,n);
```

Effects:

```
∀m(0 ≤ m < n)[f_read(gp_stack + m) = 'f_read'(e(a) + m)];
f_read(p_stack_top) = * + n;
```

```
O-function:  pop(a,n)

Purpose:  to pop n elements from the parameter stack onto location a.
Exceptions:
              write_exceptions_2(a);
              nonpositive(n);
              no_ability(gp_stack,"read");
              stack_underflow(p_stack);
              move_error(e(a),n);
              move_error(gp_stack - n-1, n);
Effects:
              ∀m(0 ≤ m < n)[f_read(e(a) + m) = 'f_read'(gp_stack - m-1)];
              ∀m(0 ≤ m < n)[f_read(gp_stack - m-1) = UNDEFINED];
              f_read(p_stack_top) = * - n;
```

```
O-function:  call(a,n)

Purpose:  to increment p_stack_bottom of the parameter stack by n;
          to transfer control to e(a)to save on the return stack the
          old p_stack_bottom of the parameter stack and the return address.

Exceptions: write_exceptions_1(a)
          nonpositive(n);
          stack_underflow(gp_stack)
          offset_error(gr_stack + 2)
          no_ability(e(a),"call");
          invalid_offset(e(a));
Effects:
          f_read(p_stack_bottom) = 'f_read'(p_stack_top) - n;
          f_read(gr_stack) = 'f_read'(program_counter) + 1;
          f_read(gr_stack + 1) = 'f_read'(p_stack_bottom);
          f_read(gr_stack_top) = * + 2;
          f_read(program_counter) = c;
```

```
O-function:  return(n)

Purpose:  to decrement p_stack_top by n;
          to restore the old p_stack_bottom and to return to the
          return address stored in the r_stack by the corresponding call.
Exceptions:
          nonpositive(n);
          stack_underflow(gp_stack);
Effects:
          f_read(p_stack_top) = 'f_read'(p_stack_bottom) + n;
          f_read(gr_stack_top) = * - 2;
          f_read(p_stack_bottom) = 'f_read'(gr_stack - 1);
          f_read(program_counter) = 'f_read'(gr_stack - 2);
```

O-function: write_into(a,i)

Purpose: to write the contents of location a into the ith element (from the top) of the current stack frame.

Exceptions:

```
        write_exceptions_1(a);
        cap_error(gp_stack);
        adr_error(gp_stack - i);
        invalid_index(i);
```

Effect:

```
        f_read(gp_stack - i) = 'f_read'(e(a));
```

O-function: write_out_of(a,i)

Purpose: to write the contents of the ith element of the current stack frame into location a.

Exceptions:

```
        write_exceptions_2(a)
        cap_error(gp_stack);
        adr_error(gp_stack - i);
        invalid_index(i);
```

Effect:

```
        f_read(e(a)) = 'f_read'(gp_stack - i);
```

A.2 Level 2. Scheduled Process Manager

Level 2 is in charge of a fixed maximum number of processes. It handles I/O devices, interrupts,process synchronization, and the system clock. All interrupts are hidden by level 2, which either handles them internally or translates them into a signal visible to the level 2 interface. Page faults, linkage faults, overflow, and addressing exceptions are all traps, which are not handled by level 2. They are all cases of hardware-detected exceptions, and are discussed in the section on error handling.

Processes at level 2 are either potentially active or blocked. A potentially active process may be running (i.e., have a processor assigned to it) or ready (i.e., eligible to be run by the dispatcher). However, this distinction (along with the scheduling algorithm) is hidden in the specifications of level 2. A blocked process may not be assigned to a processor until it is unblocked. A blocked process may be waiting to enter a monitor (or critical section), or waiting for a signal from an I/O device, the clock, or another process. A process is known at level 2 by its process_id, which can range from 1 to max_process. Process_id's which are not assigned to a process are available for assignment to processes made known to level 2. When a process is stopped, its process_id becomes available.

A process is defined by a stack segment, and a state. The stack segment corresponds to the record of activations for the process. Each activation, created by the hardware call instruction, describes an execution environment, or domain, for the process. This enables a process to operate in many possibly disjoint domains during the course of its execution. The state represents the current values of the processor registers, and must be saved explicitly when process switching takes place.

Monitors and condition variables are used for process synchronization. A monitor is a critical section of code, in which local variables pertaining to synchronization can be tested and set. Only one process can be executing in a monitor at a time. Each monitor may have a number of condition variables associated with it. A process may become blocked on a condition variable by executing a wait on that condition variable. When a process is blocked on a wait, the monitor is now free to be entered by some other process. Executing a signal on a condition variable unblocks a process waiting on that variable. Unlike semaphores, which record a V-operation when no process is waiting on a semaphore, a condition variable is not affected by a signal when no process is waiting. Thus, a signal is a no-op when there is no process waiting. (See Hoare [74].)

The system clock enables processes to be awakened after a specified time interval. This is not a real-time capability, because the specifications indicate that the process will be awakened at some time after the allotted interval is past, but there is no specification of how soon after the interval the process will actually get to run. Level 2 also allows for adding or deleting I/O devices without stopping the system. Devices can also be physically switched on and off-line by the operator.

Very little statement is made concerning how processes change their states as a result of being dispatched. One assertion is that the state of a process cannot change as long as the process is blocked. The state of a process can change in an unspecified way when a process is potentially active. In addition, a process waiting for the system clock may be awakened at some time after it is blocked.

Use of Level 2

Level 2 is used exclusively by the programs on levels 3-10. I/O drivers operate at level 3; starting and stopping of processes, and use of the system clock, is controlled by level 10; and process synchronization occurs at all levels.

Level 10 provides scheduling for the large number of processes at that level, by making processes known and unknown at level 2, according to a scheduling algorithm. A process is made known to level 2 by calling "start(cp,st)", where cp is a capability for a process and st is the state information. The new process is assigned an available process_id, and assumes a potentially active state. A *potentially active* process can be made unknown by calling "stop(cp)", where cp is the process capability. Note that a process may not be stopped when it is blocked. Furthermore, a process is not stopped until its activations have popped to a level ≥ 10. In this way level 10 cannot address any activations below itself. This would violate the correctness hierarchy of the system. For this purpose, the O-functions "call" and "return" with the ability to lock and unlock the process are provided at this level. The functions are essentially the same as those described at level 1 (in fact will be implemented using the level 1 functions), except for the locking/unlocking.

Since "stop(cp)" is delayed until process cp is active, level 10 must assume that no level 2 process becomes permanently blocked. This is ensured by allowing only levels 3-10 to use the level 2 synchronization primitives, and levels 3-10 can be proved not to have any deadlocks. This would also involve a fair scheduling algorithm, because a process can become permanently blocked with a strict priority algorithm.

A level 2 process can create a monitor by calling "initiate_monitor(n)", which creates a monitor with n condition variables and returns a capability for it. Similarly, delete_monitor(c) deletes the monitor corresponding to capability c. A monitor can be deleted only when no other process is using it or waiting to use it. "enter_monitor(c,⟨i⟩)" is called to allow process i to enter monitor c, ⟨i⟩ is an implicit parameter (i.e., not under control of the calling program). If a process is in the monitor m (i.e., "monitor_busy(m)" = TRUE), then the process is blocked and put on the waiting list for the monitor m (waiting_on_monitor(m)). A process leaving the monitor calls "exit_monitor(c,⟨i⟩)". Of course, process i must have been in the monitor to execute this function. If there are one or more processes waiting on the monitor, one of the waiting processes is unblocked and allowed to enter.

A process within a monitor can perform operations on the monitor's condition variables. "wait(m,cv)" allows process i to become blocked on condition variable cv and put on the waiting list for that condition variable "(waiting_on_condition(m,cv)". One process waiting to enter the monitor is also unblocked. "signal(m,cv)", unblocks one of the processes waiting on the condition variable cv and blocks the signaling process i, and puts it on the waiting list for the monitor m. The unblocked process gains access to the monitor m. (Implicit parameters are omitted from the discussion here.)

A process desiring to receive an interrupt calls the OV-function "receive_interrupt(c)" where c is the capability for receiving the interrupt and i is the process_id. The OV-function "receive_interrupt(c)" returns the old status of the interrupt and changes the status to a null value. If the status is null, the process is delayed.

The O-function "send_interrupt(c,i_st)" sets the status of the interrupt c to st and unblocks any process delayed on that interrupt. The O-function "clear_interrupt(c)" sets the status of the interrupt c to null.

The functions of level 2 and their specifications are given in Table A.2.

Table A.2

FUNCTIONS OF LEVEL 2

HIDDEN V-FUNCTIONS		O and OV FUNCTIONS	
{i}	= available_set		start(cp, st)^{10}
{i}	= level_2_process_set	st =	stop(cp)^{10}
st	= state(i)	c =	initiate_monitor(n)^{10}
{u}	= monitor_set		delete_monitor(c)^{10}
n	= conditions(m)		enter_monitor(c,⟨i⟩)^{10}
{i}	= waiting_on_monitor)m)		exit_monitor(c,⟨i⟩)^{10}
{i}	= waiting_on_condition(m,cv)		wait(c,cv,⟨i⟩)^{10}
{i}	= blocked		signal(c,cv,⟨i⟩)^{10}
i	= in_monitor(m)		add_to_interrupt_set(c)^{10}
b	= monitor_busy(m)		delete_from_interrupt_set(c)^{10}
i	= process_id(c)	i_st =	receive_interrupt(c,⟨i⟩)^{10}
t	= wake-up-time(i)		send_interrupt(c,i_st,⟨i⟩)^{10}
t	= clock_time		clear_interrupt(c,⟨i⟩)^{10}
{u}	= interrupt_set		put_to_sleep(t,⟨i⟩)^{10}
i_st	= i_status(u)		set_clock(t,c)^{10}
{u}	= used_set		call(c,n,⟨i⟩)^{10}
j	= lock_count(i)		return(w,⟨i⟩)^{10}

Notes: Superscript indicates highest level of accessibility.

"⟨ ⟩" denotes implicit argument supplied by the system.

LEVEL 2 PARAMETERS

b: boolean
i: integer
st: state_tuple for a process
i_st: state_tuple for an interrupt
c: capability / also c1, c2, cp/
u: unique_id / also u1, u2, up/
m: unique_id for a monitor
cv: integer / for a condition variable/
t: integer /time/
max_process: maximum number of level 2 processes
start_time: initialization time for level 2
max_monitor: maximum number of monitors
max_cv: maximum number of condition variables in a monitor
st_1: initial state of the initializing process
init_set: initial set of interrupt uids.

LEVEL 2 EXCEPTIONS

NO_SLOTS: available_set = EMPTY;

INVALID_PROCESS(i): $\neg$(1 $\leq$ i $\leq$ max_process) $\vee$ i $\in$ available_set;

NO_ABILITY(c,"a"): "a" $\notin$ abilities(c);

INVALID_CAP(cp): process_id(cp) = UNDEFINED;

ALREADY_KNOWN(cp): $\neg$ (process_id(cp) = UNDEFINED);

TOO_MANY_MONITORS: cardinality(monitor_set) $\geq$ max_monitor;

INVALID_CV(n): $\neg$ (0 $\leq$ n $\leq$ max_cv);

INVALID_MONITOR(c): u $\notin$ monitor_set;

MONITOR_BUSY(c): monitor_busy(u);

BLOCKED_PROCESSES(c): $\exists$i $\ni$ 1 $\leq$ i $\leq$ conditions(u) $\wedge$ $\neg$ (waiting_on_cv(u,i) = EMPTY);

INVALID_MONITOR_PROCESS(c,i): in_monitor(u) $\neq$ i;

INVALID_CONDITION(c,cv): $\neg$ (1 $\leq$ cv $\leq$ conditions(u));

INVALID_INTERRUPT(c): u $\notin$ interrupt_set;

INVALID_CLOCK_CAP(c): u $\neq$ clock-uid;

ALREADY_IN_INTERRUPT_SET(c): u $\in$ interrupt_set;

LEVEL 2 SPECIFICATIONS

HIDDEN V-FUNCTION: $\{i\}$ = available_set

PURPOSE: Set of process indentifiers for which no processes exist

INITIALLY: $\{i\} = \{2,\ldots,max\_process\}$

DERIVED V-FUNCTION: $\{i\}$ = level_2_process_set

PURPOSE: Set of process identifiers for which processes exist at level 2

DERIVATION: $\{i\} = \{i: 1 \leq i \leq max\_process \wedge i \notin available\_set\}$

HIDDEN V-FUNCTION: st = state(i)

PURPOSE: State(temporary location contents) for process i

INITIALLY: $\forall i[(state(i) = \text{IF } i = 1 \text{ THEN } st\_1 \text{ ELSE UNDEFINED}$

HIDDEN V-FUNCTION: $\{u\}$ = monitor_set

PURPOSE: Set of uids for valid monitors

INITIALLY: EMPTY

HIDDEN V-FUNCTION: n = conditions(m)

PURPOSE: Number of condition variables for monitor with uid m

INITIALLY: $\forall m$: UNDEFINED

HIDDEN V-FUNCTION: $\{i\}$ = waiting_on_monitor(m)

PURPOSE: Set of process identifiers of processes waiting on the monitor with uid m.

INITIALLY: $\forall m$: UNDEFINED

HIDDEN V-FUNCTION: $\{i\}$ = waiting_on_condition(m,cv)

PURPOSE: Set of process ids of processes waiting on the condition cv of the monitor m.

INITIALLY: $\forall (m,cv)$: UNDEFINED

DERIVED V-FUNCTION: $\{i\}$ = blocked

PURPOSE: Set of process ids of process waiting on some monitor or conditicn variable

DERIVATION: $\{i: \exists m(i \in waiting\_on\_monitor(m)$
$\vee \exists cv(i \in waiting\_on\_condition(m,cv)))\}$

HIDDEN V-FUNCTION: {u} = used_set

PURPOSE: Set of uids created by level 2

INITIALLY: EMPTY

HIDDEN V-FUNCTION: j = lock_count(i)

PURPOSE: The lock_count on process i, which should be zero if a process is to be stopped at level 2.

INITIALLY: ∀j: 0

HIDDEN V-FUNCTION: i = in_monitor(m)

PURPOSE: The process id of the process currently in monitor m

INITIALLY: ∀m: UNDEFINED

DERIVED V-FUNCTION: b = monitor_busy(m)

PURPOSE: Is there a process in monitor m

DERIVATION: IF (in_monitor(m) = UNDEFINED) THEN TRUE ELSE FALSE

HIDDEN V-FUNCTION: i = process_id(c)

PURPOSE: The process id assigned to process with capability c

INITIALLY: ∀c: UNDEFINED

HIDDEN V-FUNCTION: t = wake_up_time(i)

PURPOSE: Time at which process with process index i is to be woken up

INITIALLY: ∀i: UNDEFINED

HIDDEN V-FUNCTION: t = clock_time

PURPOSE: Absolute time

INITIALLY: start_time

HIDDEN V-FUNCTION: {u} = interrupt_set

PURPOSE: Set of uids for interrupts (corresponding to devices etc.)

INITIALLY: init_set

HIDDEN V-FUNCTION: i_st = i_status(u)

PURPOSE: Status of interrupt with uid u

INITIALLY: ∀u: UNDEFINED

O-FUNCTION: start(cp,st)

PURPOSE: To start the process with capability cp and state st. Process cp is made known to level 2.

```
EXCEPTIONS:  NO_SLOTS
             ALREADY_KNOWN(cp)
             NO_ABILITY(cp,"start")

EFFECTS:  CHOOSE i [i ∈ 'available_set';
                    available_set = * - i;
                    state(i) = st;
                    process id(cp) = i];
```

OV-FUNCTION: st = stop(cp)

PURPOSE: Makes process cp unknown. Returns state of the process cp

```
EXCEPTIONS:  INVALID_CAP(cp)
             NO_ABILITY(cp,"stop")

DELAY:  UNTIL process_id(cp) ∉ blocked ∧ lock_count(cp) = 0

EFFECT:  LET i = process_id(cp);
           available_set = * + i;
           state(i) = UNDEFINED;
           process_id(cp) = UNDEFINED;

VALUE:  st = 'state'(i)
```

OV-FUNCTION: c = initiate_monitor(n)

PURPOSE: To create a monitor with n condition variables and return the capability for that monitor

```
EXCEPTIONS:  TOO_MANY_MONITORS
             INVALID_CV(n)

EFFECT:  CHOOSE c ∋ uid(c) = u ∧ abilities(c) = ALL
                  ∧ [u ∉ 'used_set';
                     used_set = * + u;
                     monitor_set = * + u;
                     conditions(u) = n;
                     waiting_on_monitor(u) = EMPTY;
                     ∀i(1 ≤ i ≤ n) (waiting_on_condition(u,i) = EMPTY);
                     in_monitor(u) = UNDEFINED]

VALUE:    c
```

O-FUNCTION: delete_monitor(c)

PURPOSE: Deletes monitor with capability c, only if there is no process in it and there are no processes waiting on any condition inside the monitor.

EXCEPTIONS:
```
INVALID_MONITOR(c)
NO_ABILITY(c,"delete")
MONITOR_BUSY(c)
BLOCKED_PROCESSES(c)
```

EFFECT:
```
monitor_set = * - u;
conditions(u) = UNDEFINED;
waiting_on_monitor(u) = UNDEFINED;
∀i(1 ≤ i ≤ 'conditions'(u)): [waiting_on_condition(u,i) = UNDEFINED]
```

O-FUNCTION: enter_monitor(c,⟨i⟩)

PURPOSE: Process i (i is an implicit parameter) desires access to the monitor with capability c

EXCEPTIONS:
```
INVALID_MONITOR(c)
```

EFFECT:
```
IF ¬ 'monitor_busy'(u) THEN in_monitor(u) = i
  ELSE waiting_on_monitor(u) = * + i
```

O-FUNCTION: exit_monitor(c,⟨i⟩)

PURPOSE: Called by process i, exiting from the monitor with capability c

EXCEPTIONS:
```
INVALID_MONITOR(c)
INVALID_MONITOR_PROCESS(c,i)
```

EFFECT:
```
IF waiting_on_monitor(u) = EMPTY
  THEN in_monitor(u) = UNDEFINED
  ELSE [CHOOSE k ∋ k ∈ 'waiting_on_monitor'(u)
                  ∧ [waiting_on_monitor(u) = * - k;
                     in_monitor(u) = i]]
```

O-FUNCTION: wait(c,cv,⟨i⟩)

PURPOSE: Process i waits on condition cv of the monitor with capability c. Process i must have been inside the monitor. Access to the monitor is released

EXCEPTIONS:
```
INVALID_MONITOR(c)
INVALID_MONITOR_PROCESS(c,i)
INVALID_CONDITION(c,cv)
```

EFFECT:
```
waiting_on_condition(u,cv) = * + i;
IF waiting_on_monitor(u) = EMPTY
  THEN in_monitor(u) = UNDEFINED
  ELSE [CHOOSE k∋ k∈ 'waiting_on_monitor'(u)
                  ∧ [waiting_on_monitor(u) = * - k
                     in_monitor(u) = i]]
```

O-FUNCTION: signal(c,cv,⟨i⟩)

PURPOSE: Process i, signals condition cv inside monitor u. One of the processes, if any, waiting on that condition is unblocked and given access to the monitor u. Process i is added to waiting_on_monitor(u).

EXCEPTIONS: INVALID_MONITOR(c)
INVALID_MONITOR_PROCESS(c,i)
INVALID_CONDITION(c,cv)

```
EFFECT: IF ¬ (waiting_on_condition(u,cv) = EMPTY)
          THEN [CHOOSE k ∋ k ∈ 'waiting_on_condition(u,cv)
                         ¬ [waiting_on_condition(u,cv) = * - k ;
                            in_monitor(u) = k ;
                            waiting_on_monitor(u) = * + i]]
```

O-FUNCTION: add_to_interrupt_set(c)

PURPOSE: Increment the interrupt_set by the new uid *represented by c*

EXCEPTIONS: ALREADY_IN_INTERRUPT_SET(c)
NO_ABILITY(c,"add")

```
EFFECT: interrupt_set = * + u;
        i_status(u) = null
```

O-FUNCTION: delete_from_interrupt_set(c)

PURPOSE: To decrement *the interrupt_set* by the uid of c

EXCEPTIONS: INVALID_INTERRUPT(c)
NO_ABILITY(c,"delete")

```
DELAY: UNTIL i_status(u) = null

EFFECT: interrupt_set = * - u;
        i_status(u) = UNDEFINED
```

OV-FUNCTION: i_st = receive_interrupt(c)

PURPOSE: The working process desires to get the current status of the interrupt with capability c. The process is delayed if the current status is null. At the end of the call, the status is changed to null.

EXCEPTIONS: INVALID_INTERRUPT(c)
NO_ABILITY(c,"receive")

```
DELAY: UNTIL i_status(u) ≠ null

EFFECT: i_status(u) = null;
VALUE:  i_st = 'i_status'(u);
```

```
O-FUNCTION:  send_interrupt(c,i_st)

PURPOSE:  To set the status of the interrupt c to i_st

EXCEPTIONS:  INVALID_INTERRUPT(c)
             NO_ABILITY(c,"send")
EFFECT:      i_status(u) = i_st;
```

```
O-FUNCTION:  clear_interrupt(c,⟨i⟩)

PURPOSE:  To set the status of the interrupt c to null

EXCEPTIONS:  NO_ABILITY(c,"clear")

EFFECT:  i_status(u) = null;
```

```
O_FUNCTION:  put_to_sleep(t)

PURPOSE:  Process  wants to sleep for t units of time

DELAY:  UNTIL clock_time ≥ * + t
```

```
O-FUNCTION:  set_clock(t,c)

PURPOSE:  To set the clock_time

EXCEPTIONS:  INVALID_CLOCK_CAP(c)

EFFECT:  clock_time = t;
```

```
O-Function:  call(c,n,⟨i⟩)

Purpose:  to increment p_stack_bottom of the parameter stack by n;
          to transfer control to c; to save on the return stack the
          old p_stack_bottom of the parameter stack and the return address.

Exceptions:
          nonpositive(n);
          stack_underflow(gp_stack)
          offset_error(gr_stack + 2)
          no_ability(c,"call");

Effects:  IF lock ∈ abilities(c) THEN lock_count(i) = * + 1;
          f_read(p_stack_bottom) = 'f_read'(p_stack_top) - n;
          f_read(gr_stack) = 'f_read'(program_counter) + 1;
          f_read(gr_stack + 1) = 'f_read'(p_stack_bottom);
          f_read(gr_stack_top) = * + 2;
          f_read(program_counter) = c;
```

O-Function: return(w,⟨i⟩)

Purpose: to decrement p_stack_top by n;
to restore the old p_stack_bottom and to return to the
return address stored in the r_stack by the corresponding call.

Exceptions:

nonpositive(n);
stack_underflow(gp_stack);

Effects:

f_read(p_stack_top) = 'f_read'(p_stack_bottom) + n;
f_read(gr_stack_top) = * - 2;
f_read(p_stack_bottom) = 'f_read'(gr_stack - 1);
f_read(program_counter) = 'f_read'(gr_stack - 2);
IF unlock ε abilities(c) THEN lock_count(i) = * - 1;

A.3 Level 3: Fixed-VM Segments

Level 3 handles fixed-VM segments. A fixed-VM segment is a segment whose state is resident in the main memory. In our operating system, fixed-VM segments are the segments used by level 4 to maintain the state of the segments used by levels above level 4. Level 4 uses level 3 in the same manner as level 4 is used by higher levels. Since level 3 manages segments only for level 4, we do not need the revocation mechanism at level 3. The absence of revocation is the only difference between the external interface presented by level 3 and that presented by level 4. This level enables level 4 to address its segments as if they were in the main memory. It is expected that the data base required by level 4 will be so large that it will be uneconomical to keep it permanently in the main memory, and it will be necessary to use at least a two level store for them. Level 3 makes this invisible to level 4.

Decisions

1. Segmentation provides a two-dimensional virtual address space, with variable-sized segments for level 4.
2. Level 3 hides address translation, paging, multi-level store.
3. Level 3 provides a (small) fixed number of segments.

When a user wishes to create a segment, he calls create_seg(i), which returns a capability for a new segment of size i. All positions of the segment are zeroed out, and zero-length segments are permitted. Primitive memory operations on a segment denoted by c are performed by calling read(c,j) and write(c,j,i), where j is the displacement and i is the value to be written. Bounds checks are made for all read and write operations, by examining a segment's size (seg_size(c)). To change a segment's size, a call is made on change_seg_size(c,i). If this means enlarging the segment, then zeros fill out the new elements. When the segment is shrunk, the data in the shrunk portion goes away. Addressing within segment c is from 0 to seg_size(c)-1. A segment is deleted by calling delete_seg(c). All V-function values containing information about the segment are made undefined. There are several hidden functions, most of them related to functions already described. "seg_set" is the set of unique identifiers which refer directly to segments.

The functions externally visible at level 3 are summarized in Table A.3, and have specifications equivalent to those in level 4 (see Table A.4 in the next section).

Table A.3

Functions of Level 3

HIDDEN V-FUNCTIONS	DERIVED V-FUNCTIONS
{u} = level_3_set	b = seg_exists(c)^4
{u} = seg_set	i = seg_size(c)^4
i = h_size(u)	i = read(c,j)^4
i = h_read(u,j)	
OV- and O-FUNCTIONS	
c = create_segment(i)^4	delete_segment(c)^4
write(c,j,i)^4	change_seg_size(c,i)^4

Note: Superscript indicates the highest level of accessibility. All visible functions private to level 4.

LEVEL 3 PARAMETERS

c: capability /also c1,c2 ,cr/
u: unique_id /also u1,u2 ,uv/
b: boolean
i,j: integer
max_segs: maximum number of segments
max_size: maximum size of segments
ALL: "read", "write", "enter", "delete"

LEVEL 3 EXCEPTIONS

INVALID_SEGMENT(c): seg_exists(c) = FALSE
NO_ABILITY(c,"a"): "a" ∉ abilities(c)
INVALID_OFFSET(c,j): j ≥ seg_size(c)
TOO_MANY_SEGS: cardinality(seg_set) ≥ max_segs
INVALID_SIZE(i): ¬ (0 ≤ i ≤ max_size)

LEVEL 3 SPECIFICATIONS

HIDDEN V-FUNCTION: {u} = level_3_set

PURPOSE: Set of uids ever created through level 3

INITIALLY: EMPTY

HIDDEN V-FUNCTION: {u} = seg_set

PURPOSE: Set of uids for all segments that exist

INITIALLY: EMPTY

HIDDEN V-FUNCTION: i = h_size(u)

PURPOSE: Size of segment u

INITIALLY: UNDEFINED

HIDDEN V-FUNCTION: i = h_read(u,j)

PURPOSE: Returns the value in the jth word of the segment u

INITIALLY: UNDEFINED

DERIVED V-FUNCTION: b = seg_exists(c)

PURPOSE: External form of h_seg_exists. True iff the segment corresponding to c exists

DERIVATION: get_uid(c) ε seg_set

EXCEPTIONS: NONE

DERIVED V-FUNCTIONS: i = seg_size(c)

PURPOSE: External form of h_size. Size of segment corresponding to the capability $\bar{c}$

DERIVATION: h_size(get_uid(c))

EXCEPTIONS: INVALID_SEGMENT(c)

DERIVED V-FUNCTION: i = read(c,d)

PURPOSE: The value in the j^{th} word of the segment denoted by c

DERIVATION: h_read(get_uid(c)),j)

EXCEPTIONS: INVALID_SEGMENT(c);
NO_ABILITY(c,"read");
INVALID_OFFSET(c,j)

O-FUNCTION: change_seg_size(c,i)

PURPOSE: Change the size of segment denoted by c to i

EXCEPTIONS: INVALID_SEGMENT(c);
INVALID_SIZE(i)

LET u = get_uid(c)

EFFECTS: h_size(u) = i;
∀r(r < i)[h_read(u,r) = UNDEFINED];
∀r('h_size'(u) ≤ r < i)[h_read(u,r) = 0]

O-FUNCTION: write(c,j,i)

```
PURPOSE:  To write the value i into the $j^{th}$ word of the segment
          devoted by c

EXCEPTIONS:  INVALID_SEGMENT(c);
             NO_ABILITY(c,"write");
             INVALID_OFFSET(c,j)
LET u = get_uid(c)
EFFECTS:  h_read(u,j) = i
```

OV-FUNCTION: c = create_segment(i)

```
PURPOSE:  Returns a capability for a new segment of size i

EXCEPTIONS:  TOO_MANY_SEGS;
             INVALID_SIZE(i);

EFFECT:   CHOOSE c(LET u = get_uid(c) ∧ abilities(c) = ALL
                    $u \notin$ level_3_set;
                    level_3_set = * + u;
                    seg_set = * + u;
                    $\forall r(0 \leq r < i)$[h_read(u,r) = 0];
                    h_size(u) = i)
```

O-FUNCTION: delete_segment(c)

```
PURPOSE:  To delete the segment denoted by c

EXCEPTIONS:  INVALID_SEGMENT(c);
             NO_ABILITY(c,"delete");

LET u =  get_uid(c)

EFFECT:  seg_set = * - u;
         $\forall r(0 \leq r <$ 'h_size'(u))[h_read(u,r) = UNDEFINED];
```

A.4 Level 4: Segments and Revocation

Level 4 handles both segments and revocation. The segment part of the level provides a systemwide two-dimensional virtual memory with variable-sized segments. Since the virtual memory will be much larger than the physical memory, this level hides the possible existence of a multi-level memory structure, with or without paging. The user can address segments for which he possesses capabilities, as if the segments were in core.

Revocation provides a facility for creating a version of a capability (a revocable capability), whose power can be revoked by presenting to level 4 the revocable capability with the ability to "revoke." It is possible to create revocable copies of revocable capabilities; and it is also possible to create different revocable capabilities for the same original capability. Thus, the structure of revocable capabilities for an object can look like a tree, with many levels of indirection, and multiple revocable capabilities pointing to a single node one level closer to the object. It is possible to create revocable capabilities for segments, and for extended objects defined at higher levels than revocation.

The reason why revocation is at level 4 is two-fold. First, the address translation mechanism (performed by the hardware using level 1 tables set by level 4) must know about revocation. Since address translation is hidden above segmentation, then revocation must be at or below segmentation. Second, the revocation mechanism itself need not have its own separate knowledge of memory mapping, and its tables must be paged (because they could be quite large), so revocation must not be below segmentation either.

Following is a list of design decisions relevant to level 4:

- Revocation and segmentation are handled at t ıe same level.
- Segmentation provides a two-dimensional virtual address space, with variable-sized segments.
- Level 4 hides address translation, paging, multi-level store.
- Revocation makes it possible to create a revocable copy of a capability (called a revocable capability) whose power can be revoked by the presentation of the revocable capability with the ability to revoke.
- Revocable capabilities and object capabilities have different unique identifiers.
- Multiple revocable capabilities can correspond to the same object capability.
- A revocable capability can itself be an object capability, thus enabling many levels of indirection.
- Revocable copies can be made of capabilities for objects already defined.
- Addresses within a segment c are from 0 to seg_size(c) - 1.

The functions of level 4 are summarized in Table A.4. These are discussed next.

Segments

When a user wishes to create a segment, he calls create_seg(i), which returns a capability for a new segment of size i. All positions of the segment are zeroed out, and zero-length segments are permitted. Primitive memory operations on a segment denoted by c are performced by calling read(c,j) and write(c,j,i), where j is the displacement and i is the value to be written. Bounds checks are made for all read and write operations, by examining a segment's size (seg_size(c)). To change a segment's size, a call is made on change_seg_size(c,i). If this means enlarging the segment, then zeros fill out the new elements. When the segment is shrunk, the data in the shrunk portion goes away. Addressing within segment c is from 0 to seg_size(c)-1. A segment is deleted by calling delete_seg(c). All V-function values containing information about the segment are made undefined. There are several hidden functions, most of them related to functions already described. "virgin_set" is the set of unique identifiers which refer directly to segments.

Revocation

When it is desired to get a revocable copy of a capability c, one must call the OV-function "cr = create_revocable_cap(c)", which returns a revocable capability cr with the "revoke" ability. This capability is therefore also a revoking capability for itself and its derivatives. cr is linked to c. A revocable capability may be passed. A restricted form of it, a non-revoking revocable capability c', may also be obtained by removing the "revoke" ability. When the owner of cr wishes to revoke access via use of the capability cr, he calls "revoke(cr)" (with the ability to revoke), invalidating cr and any capabilities linked to cr (e.g., c'), but not affecting capabilities to which cr was linked (e.g., c). Multiple levels of linking are possible. This is the approach of Redell [74].

An example might be useful (Figure A.2). Suppose c is a segment capability with uid = u and Read/Write access (RW). Then "create_revocable_cap(c)" produces a revocable capability with uid = ur and access RWK (where K denotes "revoke" ability); cr is associated with c by linktuple (ur) = [ur,u]. Whenever the owner of cr wants to access the object or the uid of cr, it is exactly as if c were being presented. "chase_uid(cr)" hides the uid(ur) of cr, and returns u. Finally, "revoke(cr)" causes revocation, so that there is no further association between cr and c. Now any attempt to use cr will result in an error, as if the object corresponding to cr had never existed.

The functions of level 4 and their specifications are given in Table A.4. For historical reasons, the arguments are given as capabilities (with offset), rather than the generalized capabilities defined at level 1. In fact, it is assumed that level 4 will support generalized capabilities.

	$\begin{array}{cc} \text{uid} & \text{ac} \end{array}$
c = create_segment(i)	c $\begin{array}{\|c\|c\|}\hline u & RW \\ \hline\end{array}$
	linktuple(u) = [u]
cr = create_revocable_cap(c)	cr $\begin{array}{\|c\|c\|}\hline ur & RWK \\ \hline\end{array}$
	linktuple(ur) = [ur,u]
	chase_uid(cr) = ur
c' = restrict_ac(cr,"revoke")	c' $\begin{array}{\|c\|c\|}\hline ur & RW \\ \hline\end{array}$
revoke(cr)	chase_uid(c') = ur
	linktuple(ur) = UNDEFINED

Figure A.2 REVOCABLE CAPABILITIES AND REVOCATION

Table A.4

FUNCTIONS OF LEVEL 4

HIDDEN V-FUNCTIONS	VISIBLE V-FUNCTIONS
{u} = level_4_set	
{u} = virgin_set	
{u} = revocable_set	b = seg_exists(c)
{u} = linkset	i = seg_size(c)
[u] = linktuple(u1)	i = read(c,j)
i = h_size(u)	u = chase_uid(c)
i = h_read(u,j)	u = real_uid(c)
OV and O-FUNCTIONS	
c = create_segment(i)^7	delete_segment(c)^7
change_seg_size(c,i)	write(c,j,i)
cr = create_revocable_cap(c)^7	revoke(cr)^7

Note: Suoerscript indicates the highest level of accessibility.

LEVEL 4 PARAMETERS

c: capability /also c1,c2/,cr/
u: unique_id /also u1,u2/,ur/
b: boolean
i,j: integer
max_segs: maximum number of segments
max_size: maximum size of segments
ALL: "read","write","enter","delete","revoke"

LEVEL 4 DEFINITIONS

LET u = chase_uid(c)

LEVEL 4 EXCEPTIONS

INVALID_SEGMENT(c): seg_exists(c) = FALSE
NO_ABILITY(c,"a"): "a" $\notin$ abilities(c)
INVALID_OFFSET(c,j): j $>$ seg_size(c)
TOO_MANY_SEGS: cardinality(virgin_set) $\geq$ max_segs
INVALID_SIZE(i): $\neg$ (0 $\leq$ i $\leq$ max_size)

LEVEL 4 SPECIFICATIONS

HIDDEN V-FUNCTION: {u} = level_4_set

PURPOSE: Set of uids ever created through level 4

INITIALLY: EMPTY

HIDDEN V-FUNCTION: {u} = virgin_set

PURPOSE: Set of uids for all virgin segments

INITIALLY: EMPTY

HIDDEN V-FUNCTION: {u} = revocable_set

PURPOSE: Set of uids for revokable capabilities

INITIALLY: EMPTY

DERIVED HIDDEN V-FUNCTION: {u} = linkset

PURPOSE: Set of valid uids at this level

DERIVATION: virgin_set $\cup$ revokable_set

HIDDEN V-FUNCTION: [u] = linktuple(ul)

PURPOSE: tuple of revocable links for uid ul

INITIALLY: UNDEFINED

HIDDEN V-FUNCTION: i = h_size(u)

PURPOSE: Size of virgin segment u

INITIALLY: UNDEFINED

HIDDEN V-FUNCTION: i = h_read(u,j)

PURPOSE: Returns the value in the j^{th} word of the virgin segment u

INITIALLY: UNDEFINED

DERIVED V-FUNCTION: b = seg_exists(c)

PURPOSE: External form of h_seg_exists. True iff the segment corresponding to c exists

DERIVATION: LAST(linktuple(get_uid(c))) ∈ virgin_set

EXCEPTIONS: NONE

DERIVED V-FUNCTION: i = seg_size(c)

PURPOSE: External form of h_size. Size of segment corresponding to the capability c

DERIVATION: h_size(LAST(linktuple(get_uid(c)))

EXCEPTIONS: INVALID_SEGMENT(c)

DERIVED V-FUNCTION: i = read(c,j)

PURPOSE: The value in the j^{th} word of the segment denoted by c

DERIVATION: h_read(LAST(linktuple(get_uid(c))),j)

EXCEPTIONS: INVALID_SEGMENT(c);
NO_ABILITY(c,"read");
INVALID_OFFSET(c,j);

O-FUNCTION: change_seg_size(c,i)

PURPOSE: Change the size of segment denoted by c to i

EXCEPTIONS: INVALID_SEGMENT(c);
INVALID_SIZE(i)

EFFECTS: h_size(u) = i;
∀r(r ≤ i)[h_read(u, r) = UNDEFINED];
∀r('h_size'(u) ≤ r < i)[h_read(u, r) = 0]

O-FUNCTION: write(c,j,i)

PURPOSE: To write the value i into the j^{th} word of the segment denoted by c

EXCEPTIONS: INVALID_SEGMENT(c);
NO_ABILITY(c,"write");
INVALID_OFFSET(c,j)

EFFECTS: h_read(u, j) = i

DERIVED V-FUNCTION: u = chase_uid(c)

PURPOSE: Returns the virgin uid corresponding to the capability c

DERIVATION: IF get_uid(c) ∈ linkset THEN LAST(linktuple(get_uid(c))
ELSE get_uid(c)

EXCEPTIONS: NONE

DERIVED V-FUNCTION: u = real_uid(c)

PURPOSE: Returns the get_uid, if the revoking bit is set else
Returns the chase_uid

DERIVATION: IF "revoke" ∈ abilities(c) ∨ get_uid(c) ∉ linkset
THEN get_uid(c)
ELSE LAST(linktuple(get_uid(c)))

EXCEPTIONS: NONE

OV-FUNCTION: c = create_segment(i)

PURPOSE: Returns a capability for a new segment of sizes

EXCEPTIONS: TOO_MANY_SEGS;
INVALID_SIZE(i);

```
EFFECT: choose c(LET u = get_uid(c) ∧ abilities(c) = ALL - "revoke"
                 u ∉ level_4_set;
                 level_4_set = * + u;
                 virgin_set = * + u;
                 ∀r(0 ≤ r < i)[h_read(u,r) = 0];
                 linktuple(u) = [u];
                 h_size(u) = i)
```

O-FUNCTION: delete_segment(c)

```
PURPOSE:  To delete the segment denoted by c

EXCEPTIONS:  INVALID_SEGMENT(c);
             NO_ABILITY(c,"delete");

LET ul = LAST(linktuple(get_uid(c)))

EFFECT:  virgin_set = * - u;
         h_size(ul) = UNDEFINED;
         ∀r(0 ≤ r < 'h_size'(u)) [h_read(u, r) = UNDEFINED];
         LET z = {y | LAST(linktuple(y) = u}
         revocable_set = * - z;
         ∀x(x ∈ z)[linktuple(x) = UNDEFINED]
```

OV-FUNCTION: cr = create_revocable_cap(c)

```
PURPOSE:  Creates a revocable capability for the object represented by c

EXCEPTIONS:  NONE

EFFECT:  CHOOSE cr(LET ur = get_uid(cr) ∧
              abilities(cr) = 'abilities'(c) ∪ "revoke";
              revocable_set = * + ur;
              linktuple(ur) =
                   IF get_uid(c) ∈ linkset THEN [ur,  linktuple(u)]
                   ELSE [ur,u];
              ur ∉ level_4_set;
              level_4_set = * + ur)
```

O-FUNCTION: revoke(c)

```
PURPOSE:  Revokes the use of capability c
EXCEPTIONS:  NO_ABILITY(c,"revoke")
LET u = chase_uid(c)
EFFECT:  IF u ∉ revocable_set THEN
           [LET z = {ul | u ∈ linktuple(ul)}
            revocable_set = * - z;
            ∀ul(ul ∈ z)[linktuple(ul) = UNDEFINED]]
```

A.5 Level 5: Extended Object Manager

This level handles the creation and deletion of all objects of extended type, by which we mean all objects above level 4. It is our intention in the operating system to provide the facility for users to create new data types. We call the creator of some new type the type manager, which in turn can pass to other users the right to create objects of that type, and to manipulate those objects in specified manners. The type manager will carry out any manipulations of an extended object on behalf of the user of the object. The purpose of level 5 is to assume the burden for storing and distributing the capabilities for the objects.

One motivation for having objects of extended type follows from our goal, and that of other systems (e.g., HYDRA), to provide mechanisms that permit and facilitate the solution of special protection problems. For example, Wulf et al. [74] have devised an extended type called BIBLIOGRAPHY, and some operations that can be performed on bibliographies, namely CREATE, DELETE, PRINT, APPEND, PRINT_WITHOUT_ANNOTATIONS. Many users can be allowed to create or (when done) delete their own bibliographies. An owner of a bibliography may allow someone else to append to a bibliography that he owns, or to allow someone to print out the bibliography excluding private annotations, etc. Under no circumstances should a user be able to copy or modify the programs that perform these operations on bibliographies; those programs are private to the bibliography manager. Thus a user will be allowed to create his bibliography and attain all rights to the bibliography. He will subsequently call on the bibliography manager to manipulate the bibliography. Anybody with capabilities for a bibliography can freely distribute them to others whom he trusts.

An extended object of a type t will be composed of other objects of type t_1, t_2, ..., that are more primitive, i.e., the types t_1, t_2, ... were created before t. The most primitive object in the system is a segment, with the attendant operations on a particular segment being (as described in Appendix A.4) delete, read, write and enter. A BIBLIOGRAPHY could be composed of two objects: an ALPHABETIZED_LIST to hold the main items in order, and a segment to hold the annotations. The ALPHABETIZED_ LIST could in turn be composed of several segments. A tree of capabilities will correspond to the bibliography and its implementation. A capability cb for the bibliography will be possessed by the owner of the bibliography and by anyone who is passed this capability. Level 5 will associate with cb a capability ca for some alphabetized list and a capability s1 for a segment. Also level 5 will associate, say, two segment capabilities s2, s3 with ca. The possessor of cb will never get to possess ca or s1. Instead he will pass cb to the bibliography manager who will, after presenting cb and some manager credentials, i.e., the type manager's capability, to level 5, be returned ca and s1 for access to the alphabetized list and annotation segment. Since the alphabetized list is not primitive, the bibliography manager will need to present ca to an alphabetized list manager who will in turn attain from level 5, after presenting ca, the primitive segment capabilities s2, s3. We call ca and s1 the implementation capabilities for cb, and we call s2 and s3 the implementation capabilities for ca. Thus our system supports a hierarchy of data types, and enforces a protection of the implementation of the data type, similar to that associated with some structured programming languages, e.g., ALPHARD (Wulf [74]). The intention of this type of structure for objects, is both to hide implementation details wherever possible, and to provide facilities for flexible sharing of object-manipulating programs at any level.

The previous discussion has focused on extended objects that are created by a user. It should be noted that level 5 also manipulates the capabilities for objects at higher levels (i.e., above level 5) in the operating system. For example, level 6 is most appropriately viewed as a directory manager, for objects of type "directory." Thus level 6 will use level 5 as a repository for capabilities for all directory objects.

The notion of extended types is thus useful in the general hierarchical framework of the operating system design.

The following are the important design decisions regarding the use of level 5 and its relationship to other system levels.

1. All objects created by users will be represented by a distinguished entry capability within some directory. However, those more primitive objects which are created solely to implement some user created object are not represented within a directory. These lower level objects are "slaved" to the user object and will disappear when it is deleted. This means that the implementations for all extended objects must be created and deleted through level 5.

2. Level 5 is the only structure needed to store implementation capabilities for extended objects. The managers need not store these capabilities.

3. The *type-manager is "trusted"* by any user desiring some service for his object. Although (by virtue of (2) above) the type manager need not retain capabilities for user objects, no system function prevents him from retaining the capabilities.

4. Level 5's knowledge of an object is in terms of a tree of capabilities, the leaves of which correspond to capabilities for segments and the interior nodes of which correspond to other extended objects. The capabilities for extended objects will be of the same form as capabilities for, say, segments: c = [uid, access bits]. In the case of a capability for an extended object,the type manager interprets the access bits as the ability to perform operations on the objects. (In the case of segments, there is hardware mechanism to interpret the access bits.)

5. Operations on an extended object can be performed only by using the type manager's capability. All O- and OV-functions at levels 10 and below are assumed to be indivisible. The extended type managers above level 10 should use the monitor facilities of level 10 to assure their having exclusive access to their data when needed.

6. The creation of new types is handled by the same functions that create new objects, i.e., instance of a type. Each type is known to the system as an object of type "type."

7. Revocation of extended objects (and of course segments) is effected at level 4. A V-function at level 5 (see below) is available for determining if an object or capability is invalid because an object has been revoked, deleted, or simply does not represent an object known to the system.

   Level 5 itself can be viewed as the manager of all objects of type "type", equivalently as the manager of all of the type managers.

Access rights (abilities) meaningful to the type manager and/or object managers include:

"create", the ability to create an extended object or a type,

"delete", the ability to delete an extended object or a type, and

"manage", the ability possessed only by the manager of a type, needed to access or modify implementation capabilities.

Use of Level 5

The primary use of level 5 is in

- creating new extended types
- creating new objects of some type known to the system

- providing a type manager with the implementation capabilities associated with some object, and
- deleting objects and types.

In order to create a new type, the OV-function "create_type(t)" is called, where t is a capability with uid(t) = ut,[*] "create" ϵ abilities (t). A unique type manager's capability t' will be returned to the caller's environment with "manage" ϵ abilities(t') and "object_type(t') = tl." The "name" of the new type will be uid(t') for the lifetime of the type. An object is created in an initialized form. On the first attempted use of the object denoted by the capability c, the type manager gets an error return as the result of the exception condition UNINITIALIZED(c). The type manager then initializes the object by calling "c = create_impl_object" which creates an implementation object, with capability c. The type manager can associate any number of implementation objects with an object of its type. The implementation objects are in turn created uninitialized, and will be initialized by their type managers. Existing objects may also be used to initialize an extended-type object by calling "insert_impl_cap."

For any type t' known to the system, any user with a capability t can create an object of type t', where that uid(t) = t' and "create" ϵ abilities (t). This is achieved by calling the OV-function "create_object(t)." A new capability c for the newly created object is returned with "object_type(c)" = t' and "abilities(c)" = "all" (or the access rights that the type manager wishes to distribute). If any of these implementing objects is itself not a segment, then capabilities must be created for implementing objects of such objects, and so on, until no extended objects other than segments remain without associated implementation capabilities. An object can be deleted only by the manager of that object type. Before an object can be deleted, all objects needed for its implementation must be deleted.

[*] Recall that ut is a special integer corresponding to the uid of the manager of all type managers.

When an operation is requested on some extended object c, it is the type manager t for that object that actually carries out the operation on the implementation of the object. Thus the type manager will require the implementation capabilities for the object, which are attained by the manager by calling the function "impl_cap(c,t)."

An object c (which could also be a type) is deleted by calling the function "delete_object(c)." The effect is to cause "get_implementation_cap(t',c)," where "t' = type_object(c)" to become undefined. It is also required that any extended objects or segments that compose c also become deleted. Thus each item in the tree of capabilities becomes disassociated with any implementation capabilities. In the case of deleting a type t, no objects of that type or objects implemented using that type can be initialized. Further, when a type is deleted, all objects of that type are also deleted.

The functions of level 5 and their specifications are given in Table A.5.

Table A.5

FUNCTIONS OF LEVEL 5

	VISIBLE (DERIVED) FUNCTIONS	ABILITIES	HIDDEN FUNCTIONS
V	b = exists(c)		b = h_exists(u)
	i = object_type(c)		i = h_object_type(u)
	b = initialized(c)		b = h_initialized(u)
	[c]= impl_cap(c1,c3)		[c] = h_impl_cap(u)
			{i} = h_used_set
OV	c = create_type(c1)[7]	"create" ∈ c1	
	c = create_object(c1)[7]	"create" ∈ c1	
	c = create_impl_object	"manage" ∈ c3	
	(c1, c2,c3)	"create" ∈ c1	
O	insert_impl_cap(c1,c2,c3)	"manage" ∈ c3	
	delete_impl_cap(c1,c2,c3)	"manage" ∈ c3	
	delete_object(c1,c2)[7]	"manage" ∈ c2	

Note 7: This function restricted to levels 6 and 7.

LEVEL 5 PARAMETERS

```
b:   boolean

c:   capability   /also c1 etc/

u:   unique_id

i:   integer
max_obj:  maximum number of objects
 max_impl_cap: maximum number of impl _caps
```

LEVEL 5 DEFINITION

```
u =  chase_uid(c)   /also u1, c1; etc/

*  = all abilities present
ut = uid for type manager     /of type"type"/
```

LEVEL 5 EXCEPTIONS

```
INVALID_OBJECT(u):    h_exists(u) = FALSE
INVALID_OBJECT(c):    exists(c) = FALSE
UNINITIALIZED(u):     h_initialized(u) = FALSE
UNINITIALIZED(c):    initialized(c) = FALSE
NO_ABILITY(c,"a"):  "a" ∉ abilities(c)
INVALID_TYPE_MANAGER(ut,c):  uid(c) ≠ ut
INVALID_TYPE(c):  object_type(c) ≠ ut


INVALID_MANAGER(c1,c2):  object_type(c2) ≠ chase_uid(c1)
INVALID_IMPL_OBJECT(u1,u2):  u1 ≤ u2  /assumes monotonic uids for specification
INVALID_IMPL_CAP(c1,c2):  c2 ∉ impl_cap(c1)      simplicity only, to prevent imple-
VALID_OBJECT({c}):  ∀(c1 ∈ {c}:  exists(c)       mentation loop of A implemented
                                                 out of B implemented out of A.
INVALID_IMPL_CAPS(c):  impl_cap(c) = UNDEFINED   Alternate approaches are also
                                                 possible./
 OBJECT_NOT_CREATABLE(c):
        cardinality({c|exists(c)}) ≥ max_obj

 TOO_MANY_IMPL_CAPS(c)
        cardinality(impl_cap(c)) ≥ max_impl_cap
```

Table A.5

FUNCTIONS OF LEVEL 5

VISIBLE (DERIVED) FUNCTIONS		ABILITIES	HIDDEN FUNCTIONS
V	b = exists(c)		b = h_exists(u)
	i = object_type(c)		i = h_object_type(u)
	b = initialized(c)		b = h_initialized(u)
	[c]= impl_cap(c1,c3)		[c] = h_impl_cap(u)
			{i} = h_used_set
OV	c = create_type(c1)[7]	"create" ∈ c1	
	c = create_object(c1)[7]	"create" ∈ c1	
	c = create_impl_object	"manage" ∈ c3	
	(c1,c2,c3)	"create" ∈ c1	
O	insert_impl_cap(c1,c2,c3)	"manage" ∈ c3	
	delete_impl_cap(c1,c2,c3)	"manage" ∈ c3	
	delete_object(c1,c2)[7]	"manage" ∈ c2	

Note 7: This function restricted to levels 6 and 7.

LEVEL 5 PARAMETERS

b: boolean

c: capability /also cl, etc/

u: unique_id

i: integer

max_obj: maximum number of objects

max_impl_cap: maximum number of impl _caps

LEVEL 5 DEFINITION

u = chase_uid(c) /also ul, cl; etc/

ω = all abilities present

ut = uid for type manager /of type"type"/

LEVEL 5 EXCEPTIONS

INVALID_OBJECT(u): h_exists(u) = FALSE

INVALID_OBJECT(c): exists(c) = FALSE

UNINITIALIZED(u): h_initialized(u) = FALSE

UNINITIALIZED(c): initialized(c) = FALSE

NO_ABILITY(c,"a"): "a" ∉ abilities(c)

INVALID_TYPE_MANAGER(ut,c): uid(c) ≠ ut

INVALID_TYPE(c): object_type(c) ≠ ut

INVALID_MANAGER(cl,c2): object_type(c2) ≠ chase_uid(cl)

INVALID_IMPL_OBJECT(ul,u2): ul ≤ u2 /assumes monotonic uids for specification simplicity only, to prevent implementation loop of A implemented out of B implemented out of A. Alternate approaches are also possible./

INVALID_IMPL_CAP(cl,c2): c2 ∉ impl_cap(cl)

VALID_OBJECT({c}): ∀(cl ∈ {c}: exists(c)

INVALID_IMPL_CAPS(c): impl_cap(c) = UNDEFINED

OBJECT_NOT_CREATABLE(c):
cardinality({c|exists(c)}) ≥ max_obj

TOO_MANY_IMPL_CAPS(c)
cardinality(impl_cap(c)) ≥ max_impl_cap

LEVEL 5 SPECIFICATIONS

HIDDEN V-FUNCTION: b = h_exists(u)

PURPOSE: Does the object exist?

INITIALLY: IF u = ul THEN TRUE ELSE FALSE;

DERIVED V-FUNCTION: b = exists(c)

DERIVATION: exists(c) = h_exists(chase_uid(c))

EXCEPTIONS: NONE

HIDDEN V-FUNCTION: i = h_object_type(u)

PURPOSE: Type of the object

INITIALLY: If u = ut THEN ut ELSE UNDEFINED

DERIVED V-FUNCTION: i = object_type(c)

DERIVATION: object_type(c) = h_object_type(chase_uid(c))

EXCEPTION: INVALID_OBJECT(c)

HIDDEN V-FUNCTION: b = h_initialized(u)

PURPOSE: Is the object initialized?

INITIALLY: $\forall$u: False

DERIVED V-FUNCTION: b = initialized(c)

DERIVATION: initialized(c) = h_initialized(chase_uid(c))

EXCEPTIONS: INVALID_OBJECT(c);

HIDDEN V-FUNCTION: {i} = used_set

PURPOSE: set of uids that were ever known to this level

INITIALLY: {u_init} /initial set of uids/

HIDDEN V-FUNCTION: [c] = h_impl_cap(u)

PURPOSE: the tuple of implementation capabilities for the object u

INITIALLY: ∀u: UNDEFINED

DERIVED V-FUNCTION: [c] = impl_cap(cl,c3)

```
DERIVATION:  impl_cap(cl) = h_impl_cap(chase_uid(cl))
EXCEPTIONS:  INVALID_OBJECT(cl);
             UNINITIALIZED(cl);
             INVALID_TYPE(c3);
             NO_ABILITY(c3, "manage");
             INVALID_MANAGER(c3,cl);
```

OV-FUNCTION: c = create_type(cl)

```
PURPOSE:  to create a type manager
EXCEPTIONS:  INVALID_TYPE_MANAGER(ut,cl);
             NO_ABILITY(cl, "create");
EFFECT:  choose c ∋ uid(c) = u ∧ abilities(c) = ALL
              ∧ u ∉ 'used_set'
              ∧ [used_set = * +  u;
                h_exists(u) = TRUE;
                h_object_type(u) = ut;

                h_impl_cap(u) = UNDEFINED;
                h_initialized(u) = FALSE];
         return = c;
```

OV-FUNCTION: c = create_object(cl)

PURPOSE: to create an uninitialized object of type cl,
and to return c as its capability,
with the abilities of c = ALL

EXCEPTIONS: INVALID_TYPE(cl); /excludes segments/
NO_ABILITY(cl, "create");
OBJECT_NOT_CREATABLE(cl);

```
EFFECT:  choose c ∋ uid(c) = u ∧ abilities(c) = ALL
                   ∧ u ∉ 'used_set'
                   ∧ [used_set = * + u ;
                     h_object_type(u) = chase_uid(cl);
                     h_exists(u) = TRUE;
                     h_initialized(u) = FALSE;

                     h_impl_cap(u) = {∅}];
         return = c;
```

OV-FUNCTION: c = create_impl_object(cl,c2,c3)

PURPOSE: to create an implementation object of type c2 for the object cl. c is the capability to be returned for the created object with abilities (c) = "ALL'. c3 is the type manager's capability for the object cl.

EXCEPTIONS: INVALID_TYPE(c2); /excludes segments/
NO_ABILITY(cl,"create");
OBJECT_NOT_CREATABLE;
INVALID_OBJECT(cl);
INVALID_TYPE(c3);
NO_ABILITY(c3,"manage");
TOO_MANY_IMPL_CAPS(cl);
INVALID_MANAGER(c3,cl);
INVALID_IMPL_OBJECT(chase_uid(c2), object_type(cl)

```
EFFECTS:  choose c | uid(c) = u ∧ abilities(c) = ALL
                   ∧ u ∉ 'used_set'
                   ∧ (used_set = * + u;
                     h_object_type(u) = chase_uid(c2);
                     h_exists(u) = TRUE;
                     h_initialized(u) = FALSE;
                     h_impl_cap(u) = [EMPTY]);
                  ∧  h_impl_cap(chase_uid(c1)) = [*, c];
                     h_initialized(u1) = TRUE;
```

O-FUNCTION: insert_impl_cap(c1,c2,c3)

PURPOSE: to append c2 to the tuple impl_cap(c1). c3 is the type manager's capability for the objects of type c1.

```
EXCEPTION:  INVALID_OBJECT(c1);
            INVALID_OBJECT(c2);
            INVALID_TYPE(c3);
            NO ABILITY(c3,"manage");
            TOO MANY IMPL CAPS(c1)
            INVALID_MANAGER(c3,c1);
            INVALID_IMPL_OBJECT(object_type(c2), object_type(c1))
EFFECTS:  h_impl_cap(chase_uid(c1)) =  [*, c2];     /appends c2 to tuple/
          h_initialized(u1) = TRUE;
```

O-FUNCTION: delete_impl_cap(c1,c2,c3)

PURPOSE: to delete the implementation capability c2 from the tuple of implementation capabilities of c1. c3 is the type manager's capability for the object represented by c1. The object represented by c2 must have been deleted earlier.

```
EXCEPTIONS:  INVALID_OBJECT(c1);
             VALID_OBJECT(c2);
             INVALID_TYPE(c3);
             NO_ABILITY(c3,"manage");
             INVALID_MANAGER(c3,c1);
             INVALID_IMPL_CAP(c1,c2);
EFFECT:  impl_cap(chase_uid(c)) = [* - c2];
```

```
O-FUNCTION:  delete_object(cl,c2)

PURPOSE:  to delete the object with capability cl. The correspondence
          between the object represented capability cland its imple-
          mentation capabilities of clmust have been deleted earlier.
          c2 in the type manager's capability.
EXCEPTIONS:  INVALID_OBJECT(cl);
             NO_ABILITY(c2, "delete");
             INVALID_IMPL_CAPS(cl);
             VALID_OBJECT (imp_cap(cl));
             INVALID_TYPE(c2)
             INVALID_MANAGER(c2,cl);
EFFECT:      h_exist(chase_uid(c)) = UNDEFINED;
             h_object_type(chase_uid(c)) = UNDEFINED;
             h_initialized(chase_uid(c)) = FALSE

             h_impl_cap(chase_uid(c)) = UNDEFINED;
             IF 'h_object_type'(chase_uid(c)) = sl THEN
               [SEGEXISTS(c) = FALSE
                SEGLENGTH(c) = UNDEFINED];
```

A.6 Level 6: Directory Management

Level 6 is responsible for the creation and deletion of directory entries. The functions of level 6 are summarized in Table A.6.

There are two sets of V-functions for level 6: the hidden V-functions, which are accessible only within level 6, and the visible V-functions, which are accessible outside level 6. The visible V-functions have as an argument a capability d for the directory involved, while the hidden V-functions have the corresponding unique identifier v for that directory. These are related by the level 4 V-function v = chase_uid(d).

Each directory entry consists of a capability c and a symbolic entry name n by which that capability may be obtained. A given capability may be represented by several symbolic names, identical or different, in different directories. (The symbolic names in any one directory must of course be distinct from one another, irrespective of whether they correspond to the same capability or to different ones.) For each capability, there must be at least one directory entry that cannot be removed from its directory unless the object has already been deleted. Such an entry is said to be distinguished. (Other entries for the given capability are nondistinguished.) The existence of a distinguished entry for each object helps to prevent the existence of an object for which there is no directory entry. (See the lost-object problem, and level 7.) Creation is done by the functions "create_distinguished_entry(d,n,c)" or "create_entry(d,n,c)," depending on whether or not the entry is to be distinguished. The two functions are separate (although almost identical), because the use of "create_distinguished_entry" is restricted to level 7. Removal is done correspondingly by the function "remove_distinguished_entry(d,n)" or by "remove_entry(d,n)," and requires the "remove" ability in d. As noted above, removal of a distinguished entry is successful only if the object with the capability c has already been deleted, i.e., via "delete_object(c)" at level 5 if c is the capability for an extended-type object, or "delete_seg(c)" if c is the capability for a segment. Removal of a nondistinguished entry has no such restriction.

In addition to the set of abilities {read, write, enter, delete} relevant for segments, each type of extended object has defined a set of abilities relevant to it. (See level 5.) For directories, these abilities are {list, load, add, delete, remove, get_locks, add_locks, remove_locks}. (Note that some of these may later be made equivalent.) Table A.6 indicates which abilities are required for each of the functions at level 6. (Note that in a directory capability d, "delete" is the ability to delete the directory itself; "remove" is the ability to remove entries from the directory.)

The V-function c = get_cap(d,n,k) returns the capability for the entry named n in the directory with capability d, and succeeds (assuming no exception conditions) if and only if either the ability "load" is in d, or the *key* k fits a lock associated with the entry (or both). Each key k is a capability (for a null object); each lock is a unique identifier. The key k *fits* the *lock* u if and only if u = chase_uid(k). The access code of k is ignored. The set of locks is obtainable from the V-function "locks = get_locks(d,n)." The locks may be changed by the O-functions "add_lock(d,n,k)" and "remove_lock(d,n,w)," where w = chase_uid(k).

The O-functions "create_directory" and "delete_directory" are used exclusively by level 7. The former in turn uses "create_object" at level 5 (with type "directory"). The latter deletes a directory object, but only if it contains no distinguished entries.

The V-function {n} = dir(d) returns the set of entry names contained in the directory specified by the capability d. It requires the ability "list" in d. The V-function b = valid_dir(d) returns the value "TRUE" if and only if there exists a directory defined by the capability d. It requires the "list" ability in d. The V-function b = entry_distinguished (d,n) returns the value "true" if and only if the entry in d defined by n exists and that entry is distinguished. It requires the "list" ability in d. Finally, i = dir_size(d) returns the number of entries in d. It requires the "list" ability in d.

Each directory must itself have a distinguished entry in some (higher-level) directory (except for a directory called the root directory, with capability "root_cap"). Thus the set of all distinguished entries corresponding to directories forms a tree. An entry may be moved from one directory to another by the function "move_entry(d,n,dl,nl)." However, if a (non-) distinguished entry is moved, it must remain a (non-) distinguished entry. The "move_entry" function may also be used to rename an entry in a particular directory, e.g., "move_entry(d,n,d,nl)." Note that "move" must be specified as a primitive operation, that is, indivisible at level 6, in order to assure conservation of distinguished entries. Moving of the distinguished entry for a directory is not permitted, in order to guarantee that the directories remain tree structured. (Note: this may in fact be an unnecessary restriction, in which case it will be relaxed later.)

Table A.6

FUNCTIONS OF LEVEL 6

	Visible Function	Abilities	Hidden Functions
V	b = valid_dir(d)	"list" ε d	b = h_valid_dir(v)
	c = get_cap(d,n,k)[7]	"load" ε d /X/	c = h_get_cap(v,n)
	b = entry_distinguished(d,n)[7]	"list" ε d	b = h_entry_distinguished(v,n)
	{w} = get_locks(d,n)	"get_locks" ε d	{w} = h_get_locks(v,n)
	b = entry_exists(d,n)	"list" ε d	[c,b,{w}] = h_entry(v,n)
	{n} = dir(d)	"list" ε d	
	i = directory_size(d)	"list" ε d	
	i = lock_set_size(d,n)	"get_locks" ε d	
O, OV	create_distinguished_entry(d,n,c)[7] remove_distinguished_entry(d,n,c)[7] create_entry(d,n,c)	"add" ε d "remove" ε d "add" ε d	See other notes below
	remove_entry(d,n)	"remove" ε d	
	move_entry(d,n,dl,nl)	{"remove" ε d "add" ε dl}	
	add_locks(d,n,k)	"add_locks" ε d	
	remove_locks(d,n,w)	"remove_locks" ε d	
	d = create_directory(td)[7]	——	
	delete_directory(d,n)[7]	{"delete" ε "get_cap(d,n)"}	

Note 7: This function restricted to level 7 only.
Note X: Or an appropriate key.

LEVEL 6 PARAMETERS

b:	boolean
i:	integer
c:	capability /object/ (also cl)
d:	capability /directory/ (also dl)
n:	enter_name (also nl)
u,v,w:	unique_identifier (also ul, vl, wl)
k:	capability /key/
locks:	set unique_id

LEVEL 6 DEFINITIONS

v	-	chase_uid(d)	/also vl, dl; directory uid/
c	=	*h_get_cap(***v***,n)*	/also cl, vl, nl; cap for entry/
u	=	chase_uid(c)	/also ul, cl; uid for entry/
w	=	chase_uid(k)	
td	=	directory manager's capability	

LEVEL 6 EXCEPTIONS

DIR(v): 'hvalid_dir'(v) = TRUE;

NO_DIR(v): 'hvalid_dir'(v) = FALSE;

ENTRY(v,n): n $\in$ 'hdir'(v);

NO_ENTRY(v,n): n $\notin$ 'hdir'(v);

NO_ABILITY(d,"a"): "a" $\notin$ 'abilities'(d);

NO_KEY_MATCH(v,n,w): w $\notin$ 'hget_locks'(v,n);

NO_ACCESS(d,n,k): NO_ABILITY(d,"load") $\wedge$ NO_KEY_MATCH(v,n,w);

DIR_FULL(v): cardinality('hdir'(v)) $\geq$ maxd;

MOVE_FULL(v, vl): [v $\neq$ vl] $\wedge$ DIR_FULL(vl);

DISTINGUISHED(v,n): 'hentry_distinguished'(v,n) = TRUE;

NOT_DISTINGUISHED(v,n): 'hentry_distinguished'(v,n) = FALSE;

OBJECT_EXISTS(v,n): 'object_exists'(c) = TRUE; /level 5/

ENTRY_NOT_REMOVABLE(v,n): DISTINGUISHED(v,n);

DIR_NOT_DELETABLE(V): IF $\exists$m $\in$ 'hdir'(V) $\ni$ DISTINGUISHED(V,m) = TRUE;

NOT_KEY(k); 'get_type'(k) $\neq$ "key" $\vee$ INVALID_OBJECT(k);

NO_MORE_LOCKS(v,n): cardinality('h_get_locks'(v,n) $\geq$ locks_max;

LEVEL 6 EXCEPTION MACROS

NO_USE(d,n,"a"): NO_DIR(v);
NO_ENTRY(v,n);
NO_ABILITY(d,"a");

NO_NEW_USE(d,n,"a"): NO_DIR(v);
ENTRY(v,n);
NO_ABILITY(d,"a");

```
HIDDEN V-FUNCTION:  b = hvalid_dir(v)

PURPOSE:  Does the designated directory exist?

INITIALLY:  IF v = chase_uid(rootcap) THEN TRUE ELSE FALSE
```

```
DERIVED V-FUNCTION:  b = valid_dir(d)

/LET v = chase_uid(d), implied throughout level 6/

DERIVATION:  valid_dir(d) = hvalid_dir(v).

EXCEPTION:  NO_ABILITY(d,"list");
```

```
HIDDEN  V-FUNCTION:  c = get_cap(d,n,k)

LET w = chase_uid(k)

EXCEPTIONS:  NO_DIR(v);
             NO_ENTRY(v,n);
             NOT_KEY(k);
             NO_ACCESS(d,n,k);   /ACCESS IF "load" OR IF KEYMATCH/
INITIALLY:   HIDDEN
```

```
HIDDEN  V-FUNCTION:  b = entry_distinguished(d,n)


EXCEPTION:  NO_USE(d,n,"list");
INITIALLY:  UNDEFINED
```

```
HIDDEN   V-FUNCTION:  locks = get_locks(d,n)


EXCEPTIONS:  NO_USE(d,n,"get_locks")

INITIALLY:  UNDEFINED
```

```
DERIVED HIDDEN V-FUNCTION [c,b,{w}] = h_entry(v,n)

PURPOSE:  What is the specified entry?

DERIVATION:  tuple: [c=h_get_cap(v,n), b = h_entry_distinguished(v,n),
               {w} = h_get_locks(v,n))]
```

DERIVED V-FUNCTION b = entry_exists(d,n)

DERIVATION: IF n ∈ dir(d) THEN TRUE ELSE FALSE;

EXCEPTIONS: NO_USE(d,n,"list");

DERIVED HIDDEN V-FUNCTION: {n} = h_dir(v)

DERIVATION: {n ∋ h_get_cap(v,n) ≠ UNDEFINED}
/Note: The effects on this function are not specified,
since it is a derived function./

DERIVED V-FUNCTION: {n} = dir(d)

DERIVATION: dir(d) = hdir(v)

/Derived initial condition:
hdir(chase_uid(root_cap) = Λ /

EXCEPTION: NO_DIR(v)
NO_ABILITY(d, "list");

DERIVED V-FUNCTION: i = dir_size(d)

DERIVATION: dir_size(d) = *cardinality(dir(d))*

EXCEPTIONS: NO_DIR(v);
NO_ABILITY(d,"list");

DERIVED V-FUNCTION: i = lock_set_size(d,n)

DERIVATION: cardinality(get_locks(d,n))

EXCEPTIONS: NO_USE(d,n,"get_locks").

O-FUNCTION: create_distinguished_entry(d,n,c)

/h_dir(v) = * + n/

PURPOSE: to create a distinguished entry with the given entry name in the specified directory, for an object with the given capability

EXCEPTIONS: NO_NEW_USE(d,n,"add")
DIR_FULL(v);

EFFECTS: h_entry(v,n) = (c, TRUE, EMPTY) /capability, distinguished, locks/

```
O-FUNCTION: remove_distinguished_entry(d,n)

PURPOSE:  to remove a distinguished entry, but only if the
          corresponding object has been ddleted.

EXCEPTIONS:  NO_use(d,n, "remove")

             NOT_DISTINGUISHED(v,n)

EFFECTS:  h_entry(v,n) = UNDEFINED


O-FUNCTION:  create_entry(d,n,c)                    /h_dir( ) = * + n/

PURPOSE:  to create a nondistinguished entry

EXCEPTIONS:  NO_NEW_USE(d,n,"add")
             DIR_FULL(v);

EFFECTS:     h_entry(v,n) = (c, FALSE, EMPTY)


O_FUNCTION:  remove_entry(d,n)                      /h_dir( ) = * - n/

PURPOSE:  to remove the named nondistinguished entry from the specified directory

EXCEPTIONS:  NO_USE(d,n,"remove");

             DISTINGUISHED(v,n);

EFFECTS:     h_entry(v,n) = UNDEFINED


O-FUNCTION:  move_entry(d,n,dl,nl)               /Move or rename/

PURPOSE:  to move an entry from one directory to another (or to rename an
          entry in the same directory)

EXCEPTIONS:  NO_USE(d,n,"load" ∧ "remove");
             NO_NEW_USE(dl,nl,"add")
             DIR(u)     /directories not movable in order to maintain
                          tree structure/
             MOVE_FULL(v,vl);

EFFECTS:     IF (v = vl) ∧ (n = nl) THEN EMPTY ELSE

                  [h_entry(v,n) = UNDEFINED;
                   h_entry(vl,nl) = *(v,n)];


OV-FUNCTION:  d = create_directory

EXCEPTIONS:  OBJECT_NOT_CREATABLE                /level 5/

EFFECTS:  CHOOSE d ∍ [h_valid_dir(v) = TRUE;
                      abilities(d) = ALL_
```

```
O-FUNCTION:  delete_directory(d,n)

PURPOSE:  to delete a directory, but only if (d,n) is a distinguished
          entry for the directory, which itself contains no distinguished
          entries.  (The directory d remains unchanged.  Removal of the
          entry (d,n) is the subsequent responsibility of level 7.)

EXCEPTIONS:  NO_DIR(v);
             NO_ENTRY [illegible]
             NO_ABILITY(c,"delete");
             NOT_DISTINGUISHED(v,n);     /entry for the directory to be deleted/
             NO_DIR(u);                  /v is uid of dir to be deleted/
             DIR_NOT_DELETABLE(u);       /v contains a distinguished entry/

EFFECTS:  hvalid_dir(u) = FALSE;
       ∀m [h_entry(u,m) = UNDEFINED];
```

```
O-FUNCTION:  add_lock(d,n,k)

PURPOSE:  to add a lock to the lock set
LET w = chase_uid(k);

EXCEPTIONS:  NO_USE(d,n,"add_lock")
             NO_MORE_LOCKS(d,n)

EFFECTS:  h_entry(v,n) = (*,*,* + w);
```

```
O-FUNCTION:  remove_lock(d,n,w)

PURPOSE:  to remove a lock from the lock set.

EXCEPTIONS:  NO_USE(d,n,"remove_lock");
             NO_KEY_MATCH(v,n,w);

EFFECTS:  h_entry(v,n) = (*,*,* - w);
```

A.7 Level 7: User Object Manager

Level 7 exists solely to guarantee that no objects can be lost in the course of normal operation. (An object created at level 7 or above would be considered lost if there were no capability for it in any directory.) The operations at level 7 are given in Table A.7. The operation "create_object(d,n,t)" creates an object of the appropriate type t and creates a distinguished directory entry n in directory d for which "c = get_cap(n,d)." The operation "delete_object(d,n)" destroys the object, and removes its distinguished directory entry. The comparable operations for segments and directories are similar to this pair of functions, and are separate primarily because they are implemented quite differently. (They could in fact be lumped under the more general first pair of functions by treating segments and directories as ordinary objects.)

Level 7 has no state information beyond that of the lower levels. That is, while it provides procedure abstraction, it provides no data abstraction. Thus level 7 is more understandable if "specified" in terms of the abstract implementations (and the implicit effects on lower-level V-functions). It is straightforward to replace these abstract implementations by specifications, first creating V-functions at level 7 identical to the relevant lower-level V-functions, then inserting the lower-level specifications for the relevant lower-level O-functions, and then adding to the specifications any consistency requirements on internal variables (e.g., the invariance of the capability c in "create_object"). Specifications for level 7 functions are given in Table A.7.

Functions to create a revocable capability and to revoke are also included at level 7, reflecting the corresponding level 4 functions, but assuring that directory entries are also maintained for the revoking form of all revocable capabilities. Thus revoking capabilities (with the ability "revoke") are also objects that cannot be lost. This is of value in simplifying level 4.

Table A.7

LEVEL 7 FUNCTIONS

OV-FUNCTIONS	O-FUNCTIONS
c = create_object(d,n,t)	delete_object(d,n)
s = create_segment(d,n,i)	delete_segment(d,n)
dl = create_directory(d,n)	delete_directory(d,n)
ck = create_revocable_cap(dk,nk,c)	revoke(ck)

LEVEL 7 PARAMETERS

d,dk:	capability	/directory/
n,nk:	entry_name	
t,td:	capability	/type manager/
c:	capability	
key:	capability	/with no implementation/

LEVEL 7 DEFINITIONS

v = chase_uid(d)	/also vk,dk; level 5/
c = get_cap(d,n)	/level 6/
u = chase_uid(c)	/level 5/

LEVEL 7 EXCEPTIONS

Appropriate exceptions are defined in levels 6, 5 or 4, as indicated.

```
O-FUNCTION:  c = create_object(d,n,t)

PURPOSE:  to create catalogued object of the specified type

EXCEPTIONS:  INVALID_TYPE(t,"seg");                       /level 5/
             INVALID_TYPE(t,"dir");                       /level 5/
             OBJECT_NOT_CREATABLE;                        /level 5/
             NO_TYPE(t);                                  /level 5/
             NO_ABILITY(t, "create");                     /level 5/
             NO_NEW_USE(d,n,"add");                       /level 6/
             DIR_FULL(v)                                  /level 6/

EFFECTS  (abstract implementation):                       /level 5/

      BEGIN:
        c = create_object(t);                             /level 5/
        create_distinguished_entry(d,n,c);                /level 6/
      END;

O-FUNCTION:  delete_object(d,n)

PURPOSE:  to delete an object and remove its distinguished entry

EXCEPTIONS:  INVALID_TYPE(c,"dir");                       /level 5/
             INVALID_TYPE(c,"seg");                       /level 5/
             NO_OBJECT(u);                                /level 5/
             NOT_DISTINGUISHED(v,n);                      /level 6/
             NO_ABILITY(c,"delete");                      /level 5/
             NO_USE(d,n,"remove")                         /level 6/

EFFECTS  (abstract implementation):
      BEGIN:
             [delete_object(c);                           /level 5/
              remove_distinguished_entry(d,n)];           /level 6/

O-FUNCTION:  s = create_segment(d,n,i)

PURPOSE:  to create a catalogued segment of length i

EXCEPTIONS:  SEGMENT_NOT_CREATABLE;                       /level 4/
             INVALID_SIZE(i)                              /level 4/
             NO_NEW_USE(d,n,"add");                       /level 6/
             DIR_FULL(v)                                  /level 6/

EFFECTS  (abstract implementation):
      BEGIN:
        s = create_segment(i);                            /level 4/
            create_distinguished_entry(d,n,s);            /level 6/

      END;
```

O-FUNCTION: delete_segment(d,n)

PURPOSE: to delete a segment and remove its distinguished directory entry;

```
EXCEPTIONS:  NO SEG(u)                                /level 4/
             NOT_DISTINGUISHED(v,n)                   /level 6/

             NO_ABILITY(c,"delete")                   /level 4/
             NO_USE(d,n,"remove")                     /level 6/


EFFECTS: (abstract implementation):
     BEGIN:
       delete_segment(c);                             /level 4/
       remove_distinguished_entry(d,n);               /level 6/
     END;
```

O-FUNCTION: dl=create_directory(d,n)

PURPOSE: to create a new directory and catalogue it.

```
EXCEPTIONS:  OBJECT_NOT_CREATABLE                     /level 5/
             NO_NEW_USE(d,n,"add");                   /level 6/
             DIR_FULL(v)                              /level 6/

EFFECTS:  (abstract implementation)
     BEGIN:
      dl = create_object( td);                            /level 5/
           create_distinguished_entry(d,n,dl);            /level 6/
     END;
```

O-FUNCTION: delete_directory(d,n)

PURPOSE: to delete a directory and remove the distinguished entry for it.

```
EXCEPTIONS:  NO DIR(u);                     /level 6/
             NOT_DISTINGUISHED(v,n);        /level 6/
             DIR_NOT_DELETABLE(u);          /dir contains distinguished entries/
             NO_ABILITY(c,"delete")         /level 5/
             NO_USE(d,n,"remove");          /level 6/

EFFECTS:  (abstract implementation)

     BEGIN:
       delete_directory(d,n);                             /level 6/
       remove_distinguished_entry(d,n);                   /level 6/
     END;
```

```
OV-FUNCTION:  ck = create_revocable_cap(dk,nk,c)

PURPOSE:  to create a revoking and revocable capability and catalogue it

EXCEPTIONS:  NO_NEW_USE(dk,nk,"add")                       /level 6/

EFFECTS:  (abstract implementation)
     BEGIN:
       cr = create_revocable_cap(c);                    /level 4/
       create_distinguished_entry(dk,nk,cr);            /level 6/
```

```
O-FUNCTION:  revoke(dk,nk,key)

PURPOSE:  to revoke a revocable capability

LET ck = get_cap(dk,nk)

LET ukey = chase_uid(key)

EXCEPTIONS:  NO_USE(dk,nk,"remove");                     /level 6/
             NOT_KEY(key);                               /level 6/
             NO_KEY_MATCH(vk,nk,ukey);                   /level 6/
             NO_ABILITY(ck,"revoke");                    /level 4/
             NO_ACCESS(dk,nk,key);                       /level 6/
             NOT_DISTINGUISHED(dk,nk);                   /level 6/

EFFECTS:  (abstract implementation)
     BEGIN:
       ck = get_cap(dk,nk);                             /level 6/
            revoke(ck);                                 /level 4/
            remove_entry(dk,nk);                        /level 6/
     END;
```

A.8 Level 8: Linkage Maintainer

Level 8 is responsible for the management of linkage sections whose use provides efficient normal access to objects while also facilitating symbolic initial access. For each object (e.g., procedure) requiring symbolic access to other objects (e.g., other procedures, or data), a linkage template is created (e.g., by a compiler). The linkage template is then considered to be pure data, and does not change during execution. Each entry in a linkage template may contain a symbolic name (to be linked eventually to a capability) or a capability (if prelinking is desired), or both. When the object is actually to be used, a linkage section is derived from the linkage template. (Note that a linkage section need not be created if the linkage template is completely prelinked.) For any intended symbolic access for which no linkage exists in the linkage section (causing a linkage fault), the linker (level 9) finds a suitable object. The linker is driven by a search strategy that employs the directories at level 6 and that uses the level 8 primitives to maintain the relevant linkage sections. The capability for the object obtained from the directory system is then implanted in the appropriate entry of the linkage section. This action is called dynamic linkage, being done as a part of the first attempt to use an unlinked entry in a linkage section. (See level 9.) On subsequent access, the desired capability is obtained in a single instruction via indirection through the appropriate linkage section entry. In principle, the use of linkage here is similar to Multics (Janson [74]).

Level 8 is thus responsible for the creation and management of linkage templates, and for the creation and management of linkage sections. The index of each linkage section entry is visible onl; at level 9. The functions available at level 8 are summarized in Table A.8, and are divided into two parts, those relevant to linkage templates, and those relevant to linkage sections.

Table A8

FUNCTIONS OF LEVEL 8
(Linkage Maintainer)

	V-FUNCTIONS	OV,O-FUNCTIONS
OPERATIONS ON LINKAGE TEMPLATES	b = ℓtemplate_exists(ℓt)	ℓt = create_ℓtemplate(tℓ,j)
	j = get_ℓsize(ℓt)	delete_ℓtemplate(ℓt)
	b = ℓname_defined(ℓt,i)	define_ℓname(ℓt,i,n)
	n = get_ℓname(ℓt,i)	prelink(ℓt,i,c)
	b = prelink_exists(ℓt,i)	
	c = get_prelink(ℓt,i)	
OPERATIONS ON LINKAGE SECTIONS	b = ℓsection_exists(ℓ)[9]	ℓ = create_ℓsection(ℓt)[9]
	ℓt = get_templcap(ℓ)[9]	delete_ℓsection(ℓ)[9]
	b = link_exists(ℓ,i)[9]	link(ℓ,i,c)[9]
	c = get_link(ℓ,i)[9]	unlink(ℓ,i)[9]

Note 9: Restricted to level 9

LEVEL 8 PARAMETERS

b:	boolean	
c:	capability	/object/
n:	name	/linkage name/
ℓ:	capability	/linkage section/
ℓt:	capability	/linkage template/
i:	integer	/index into ℓ or ℓt/
j:	integer	/size of ℓ or ℓt/
tℓ:	capability	/for creating linkage template/

LEVEL 8 DEFINITIONS

ℓ = get_template_cap(ℓt)

ℓt_entry(ℓt,i) = (ℓname_exists(ℓt,i), get_ℓname(ℓt,i),
prelink_exists(ℓt,i), get_prelink(ℓt,i))

LEVEL 8 EXCEPTIONS

NO_ABILITY(c,"a"): "a" ∉ 'abilities'(c);

NO_TEMPLATE(ℓt): 'ℓtemplate_exists'(ℓt) = FALSE;

INVALID_INDEX(ℓt,i): (i < 0) ∨ (i ≥ 'get_ℓsize'(ℓt));

UNDEFINED_NAME(ℓt,i): 'ℓname_defined'(ℓt,i) = FALSE;

DEFINED_NAME(ℓt,i): 'ℓname_defined'(ℓt,i) = TRUE;

EXCESSIVE_SIZE(j): j > ℓsize_max;

PRELINK(ℓt,i): 'prelink_exists"(ℓt,i) = TRUE;

NO_SECTION(ℓ): 'ℓsection_exists'(ℓ) = FALSE;

NO_LINK(ℓ,i): 'link_exists'(ℓ,i) = FALSE;

LINK_EXISTS(ℓ,i): 'link_exists'(ℓ,i) = TRUE;

LEVEL 8 SPECIFICATIONS - LINKAGE TEMPLATES

V-FUNCTION: b = ℓtemplate - exists(ℓt)

PURPOSE: Does the designated linkage template exist?

INITIALLY: UNDEFINED

EXCEPTION: NO_ABILITY(ℓt,"read");

V-FUNCTION: j = get_ℓsize(ℓt)

PURPOSE: How many names are linkable in the designated linkage template?

INITIALLY: UNDEFINED

EXCEPTION: NO_TEMPLATE(ℓt);
NO_ABILITY(ℓt,"read");

V-FUNCTION: b = ℓname_defined(ℓt,i)

PURPOSE: Is the name defined for the ith entry in the given linkage template?

INITIALLY: UNDEFINED

EXCEPTIONS: INVALID_INDEX(ℓt,i);
NO_ABILITY(ℓt,"read");

V-FUNCTION: n = get_ℓname(ℓt,i)

PURPOSE: To obtain the name of the ith entry in the given linkage template.

INITIALLY: UNDEFINED

EXCEPTIONS: UNDEFINED_NAME(ℓt,i);
ABILITY(ℓt,"read");

V-FUNCTION: b = prelink_exists(ℓt,i)

PURPOSE: Is the specified entry in the given linkage template prelinked?

INITIALLY: UNDEFINED

EXCEPTIONS: INVALID_INDEX(ℓt,i);
NO_ABILITY(ℓt,"read");

V-FUNCTION: c = get_prelink(ℓt,i)

PURPOSE: What is the capability corresponding to the desired entry of the given linkage template?

INITIALLY: UNDEFINED

EXCEPTIONS: NO_PRFLINK(ℓt,i);
NO_ABILITY(ℓt,"read");

OV-FUNCTION: ℓt = create_ℓtemplate(j,tℓ)

PURPOSE: To create a linkage template.

EXCEPTIONS: EXCESSIVE_SIZE(j);
NO_ABILITY(tℓ,"create");

EFFECTS: ℓtemplate_exists(ℓt) = TRUE;
get_ℓsize(ℓt) = j;
ℓname_exists(ℓt,i) =
IF 0 ≤ i ≤ j THEN FALSE ELSE UNDEFINED;
get_ℓname(ℓt,i) = UNDEFINED;
prelink_exists(ℓt,i) =
IF 0 ≤ i ≤ j THEN FALSE ELSE UNDEFINED;
get_prelink(ℓt,i) = UNDEFINED

O-FUNCTION: delete_ℓtemplate(ℓt)

PURPOSE: To delete a linkage template.

EXCEPTIONS: NO_TEMPLATE(ℓt);
NO_ABILITY(ℓt,"delete");

EFFECTS: ℓtemplate_exists(ℓt) = FALSE
get_ℓsize(ℓt) = UNDEFINED;
∀i[ℓt_entry(ℓt,i) = UNDEFINED];

O-FUNCTION: define_ℓname(ℓt,i,n)

PURPOSE: To establish an unlinked entry in an ℓtemplate for a given name.

EXCEPTIONS: NO_TEMPLATE(ℓt);
INVALID_INDEX(i);
DEFINED_NAME(ℓt,i);
NO_ABILITY(ℓt,"add");

EFFECTS: ℓname_exists(ℓt,i) = TRUE;
get_ℓname(ℓt,i) = n;

O-FUNCTION: prelink(ℓt,i,c)

PURPOSE: To establish a link in the given linkage template.

EXCEPTIONS: NO_TEMPLATE(ℓt);
INVALID_INDEX(i);
PRELINK(ℓt,i);
NO_ABILITY(ℓt,"link");

EFFECTS: prelink_exists(ℓt,i) = TRUE;
get_prelink(ℓt,i) = c;

V-FUNCTION: b = ℓsection_exists(ℓ)

PURPOSE: Does the given linkage section exist?

INITIALLY: UNDEFINED

EXCEPTIONS: [illegible]

V-FUNCTION: ℓt = get_templcap(ℓ)

PURPOSE: What is the template capability corresponding to the given linkage section?

INITIALLY: UNDEFINED

EXCEPTIONS: NO_SECTION(ℓ);

V-FUNCTION: b = link_exists(ℓ,i)

PURPOSE: is the specified entry in the given linkage section linked?

INITIALLY: UNDEFINED

EXCEPTIONS: NO_SECTION(ℓ)
INVALID_INDEX(i)
NO_ABILITIES(ℓ,"load");

V-FUNCTION: c = get_link(ℓ,i)

PURPOSE: To obtain the capability corresponding to the desired entry of the given linkage section.

INITIALLY: UNDEFINED

EXCEPTIONS: NO_LINK(ℓ,i)
NO_ABILITY(ℓ,"load");

OV-FUNCTION: ℓ = create_ℓsection(ℓt)

PURPOSE: to create a linkage section from the given template.

EXCEPTIONS:
NO_TEMPLATE(ℓt);
NO_ABILITY(ℓt,"?");

EFFECTS:
ℓsection_exists(ℓ) = TRUE;
get_template_cap(ℓt) = ℓ;
∀i[link_exists(ℓ,i) = prelink_exists(ℓt,i)];
∀i[get_link(ℓ,i) = get_prelink(ℓt,i)];

O-FUNCTION: delete_ℓsection(ℓ)

PURPOSE: To delete a given linkage section.

EXCEPTIONS:
NO_SECTION(ℓ);
NO_ABILITY(ℓ,"?");

EFFECTS:
ℓsection_exists(ℓ) = FALSE;
get_template_cap(ℓ) = UNDEFINED;
∀i[link_exists(ℓ,i) = UNDEFINED];
∀i[get_link(ℓ,i) = UNDEFINED];

O-FUNCTION: link(ℓ,i,c)

PURPOSE: To link a given entry to a given capability.

EXCEPTIONS:
NO_SECTION(ℓ);
INVALID_INDEX(ℓt,i);
LINK(ℓ,i);
NO_ABILITY(ℓt,"link");

EFFECTS:
link_exists(ℓ,i) = TRUE;
get_link(ℓ,i) = c;

O-FUNCTION: unlink(ℓ,i)

PURPOSE: To unlink the given entry

EXCEPTIONS:
NO_SECTION(ℓ);
NO_LINK(ℓ,i);
PRELINK(ℓt,i);
ABILITY(ℓ,"unlink");

EFFECTS:
link_exists(ℓ,i) = UNDEFINED
get_link(ℓ,i) = UNDEFINED

A.9 Level 9: Linkage Manager (Linker)

Level 9 is responsible for the management of the linkage tables maintained by level 8. It adds no new state information to the system beyond what is already part of level 8. As noted in the previous section, there are two types of linkage, dynamic and static linkage. Static linkage may be obtained by using prelink (ℓt,i,c) for global linkage, affecting all symbolic accesses, or by using link (ℓ,i,c) for local linkage relevant only within a particular process. Dynamic linkage is discussed here.

Dynamic Linkage

A linkage fault occurs when an attempt to use a linkage section entry results in no capability being obtained--i.e., for a given linkage section ℓ and entry number i, when linkage_exists (ℓ,i) = FALSE. In response to the linkage fault, level 9 is invoked. The directories dj = {search_dir(j)}, j = 1 to j = search_length, are searched in order, until a directory (with capability d) is found having an entry with the corresponding symbolic entry name n = get_ℓname(ℓt,i), where ℓt = get_template_cap(ℓ). From this directory entry, the capability c = get_cap(d,n) is obtained, and entered into the given linkage section via link(ℓ,i,c). Thus the abstract implementation for the dynamic linker is as in Table A.9. The function "search_and_link" is internal to level 9.

Table A.9

FUNCTION OF LEVEL 9
(Linker)

```
O-FUNCTION:  search_and_link(ℓ,i)                         /linker/

PURPOSE:  to find an appropriate directory entry and link its capability.

LET ℓt = get_template_cap(ℓ);
     n = get_ℓname(ℓt,i);                                 /desired name to be linked/
    dj = search_list(j);                                  /jth dir in search/
    cj = get_cap(dj,n);                                   /desired capability; level 6/

EXCEPTIONS:  NO_SECTION(ℓ);
             INVALID_INDEX(ℓt,i);
             LINK_EXISTS(ℓ,i);
             NO_ABILITY(ℓt,"link");
             NO_USE(dj,n,"list");

EFFECTS:  (abstract implementation)

     FOR j = 1 TO j = search_length DO
       BEGIN:
         IF entry_exists(dj,n)THEN[link(ℓ,i,cj);EXIT]
       END;

/EXPANDED EFFECTS, for comparison:
     FOP j = 1 TO search_length DO
       BEGIN:
         IF entry_exists(search_list(j),get_ℓname(get_template_cap(j),i))
         THEN [link(ℓ,i,get_cap(search_dir(j),get_ℓname
              (get_template_cap(ℓ),i)));
              EXIT]
       END;/
```

/Primitives to modify the search list are omitted for simplicity. Search_list may be established by default for each user, as in Multics, e.g., working directory, process directories, libraries,.../

AD-A088 601 STANFORD RESEARCH INST MENLO PARK CALIF F/G 15/3
A PROVABLY SECURE OPERATING SYSTEM.(U)
JUN 75 P G NEUMANN, L ROBINSON, K N LEVITT DAAB03-73-C-1454

UNCLASSIFIED NL

4 OF 4
AD A
088601

END
DATE
FILMED
10-80
DTIC

A.10 Level 10: Scheduling

Level 10 has the responsibility for managing processes. It provides the O-functions for creating, deleting, starting and stopping processes. It also provides facilities for interprocess communication and for creating protected environments within a process.

Each process created has a unique identifier. An initial state has to be specified when a process is created. When a process is created, it is added to the set of known processes. It is made unknown when it is deleted. To be eligible to be run, a process has to be started after it has been added to the set of schedulable processes. A schedulable process can be in one of three states: scheduled, i.e., known to level 2; ready-to-be scheduled, i.e., can be made known to level 2; and blocked, i.e., waiting on some condition. The distinction between the first two states is invisible above level 10. A process which is not blocked may be stopped, at which point it is deleted from the set of schedulable processes. A process may be deleted at any time. A deleted process is also made unschedulable.

A process consists of a sequence of nested environments of depth at least 1. The topmost environment of a process is called the current environment. An environment is a tuple of values.

A process normally accesses the current environment of its own set of environments. A process may access or change the environments of other stopped processes by presenting appropriate capabilities. An environment is created by the O-function "call" and deleted by the O-function "return." Entries may be added or deleted from the environment with a last-in-first-out discipline, by using the O-functions "push" and "pop." The contents of any entry in the environment may be read out of or written into by the O-functions "write-out-of" and "write-into."

The interprocess communication facilities, monitors and condition variables, are identical to those provided at level 2.

The functions of level 10 and their specifications are given in Table A.10.

Table A.10

FUNCTIONS OF LEVEL 10

HIDDEN V-FUNCTIONS	OV AND O-FUNCTIONS
{u} = h_known_process_set	c = create_process(st)
{u} = h_schedulable_process_set	start(c)
st = h_state(up)	stop(c)
c = h_program_counter_cap(up)	delete_process(c)
i = h_program_counter_offset(up)	call(f,n,⟨ce⟩)
c = h_current_env_cap((up⟩	return(n,⟨ce⟩,⟨up⟩)
b = h_env_cap(u,up)	push(f,n,⟨ce⟩)
f = h_entry_point_env(u)	pop(f,n,⟨ce⟩
f = h_return_address_env(u)	write_out_of_env(f,i,⟨ce⟩)
c = h_previous_env_cap(u)	write_into_env(f,i,⟨ce⟩)
i = h_env_length(u)	set_upper-bound(n,⟨ce⟩)
j = h_read_env(i,u)	write_env(f,i,c)
{u} = monitor_set	truncate_env(c,n)
n = conditions(m)	append_env(f,c)
{u} = waiting_on_monitor(m)	c = create_monitor(cm,n)
{u} = waiting_on_condition(m,cv)	delete_monitor(c)
{u} = blocked	enter_monitor(c,⟨cp⟩)
{u} = in_monitor(m)	exit_monitor(c,⟨cp⟩)
b = monitor_busy(m)	wait(c,cv,⟨cp⟩) signal(c,cv,⟨cp⟩
VISIBLE V-FUNCTION	
j = read_env(i,c)	

LEVEL 10 PARAMETERS

b:	boolean	
i:	integer	
j:	integer	
n:	integer	
c:	capability	/cl /
f:	offset_capability	
u:	unique_id	/ul,ue,up/
m:	unique_id for a parameter	
cv:	integer	/for a condition variable/

max_monitor: maximum number of monitors
max_cv: maximum number of condition variables in a monitor

ce:	capability	/current environment/
cm:	capability	/monitor creation/
cp:	capability	/process/

LEVEL 10 DEFINITIONS

u = chase_uid(c)	/also up, cp/

EXCEPTIONS FOR LEVEL 10

```
INVALID_c(c,cp):  h_env_cap(chase_uid(c),chase_uid(cp))
INVALID_i(i,c):  ¬(0 ≤ i ≤ h_env_length(chase_uid(c))
INVALID_n(n):  n ≤ 0;
NO_SPACE:  cardinality(known_process_set) ≥ max_known_processes
UNKNOWN(c):  chase_uid(c) ∉ known_process_set
NO_ABILITY(c,"a"):  "a" ∉ abilities(c)
SCHEDULABLE(c):  chase_uid(c) ∈ schedulable_process_set
UNSCHEDULABLE(c):  chase_uid(c) ∉ schedulable_process_set
NO_ABILITY(f,"a"):  "a" ∉ abilities(f)
INVALID_OFFSET(f,i):  offset(f) ≠ 0;
OFFSET_ERROR(f):  displacement(f) > seg_length(f)
OVERFLOW(f):  offset(f) > max_segment_length
OVERFLOW_ENV(c,i):  h_env_length(u) > max_segment_length - i;
INVALID_ENV_CAP(c):  ∄ up ∋ [h_env_cap(chase_uid(c),up) ∧
                              [up ∈ schedulable_process_set ∨
                               up ∉ known_process_set]]


TOO_MANY_MONITORS:  cardinality(monitor_set) ≥ max_monitor;
INVALID_CV(n): ¬ (0 ≤ n ≤ max_cv);
INVALID_MONITOR(c):  u ∉ monitor_set;
MONITOR_BUSY(c):  monitor_busy(u);
BLOCKED_PROCESSES(c):  ∃i ∋ 1 ≤ i ≤ conditions(u) ∧ ¬ (waiting_on_cv(u,i) = EMPTY);
INVALID_MONITOR_PROCESS(c,cp): in_monitor(u) ≠ up;
INVALID_CONDITION(c,cv): ¬ (1 ≤ cv ≤ conditions(u));
```

HIDDEN V-FUNCTION: {u} = h_known_process_set

PURPOSE: Set of process uids known to this level

INITIALLY: Λ

HIDDEN V-FUNCTION: {u} = h_schedulable_process_set

PURPOSE: Set of process uids that are schedulable

INITIALLY: Λ

HIDDEN V-FUNCTION: st = h_state(up)

PURPOSE: The state of process up, up $\in$ known_process_set

INITIALLY: $\forall$ up: UNDEFINED

HIDDEN V-FUNCTION: c = h_program_counter_cap(up)

PURPOSE: The capability part of the contents of program counter of process up

INITIALLY: $\forall$ up: UNDEFINED

HIDDEN V-FUNCTION: i = h_program_counter_offset(up)

PURPOSE: The offset part of the contents of program counter of process up

INITIALLY: $\forall$ up: UNDEFINED

HIDDEN V-FUNCTION: i = h_env_length(u)

PURPOSE: What is the length of the environment identified by uid u

INITIALLY: $\forall$(u): UNDEFINED

HIDDEN V-FUNCTION: j = h_read_env(i,u)

PURPOSE: To get the value of the i^{th} word in the environment identified by uid u

INITIALLY: $\forall$(i,u): UNDEFINED

DERIVED V-FUNCTION: j = read_env(i,c)

PURPOSE: To get the value of the ith word in the environment c

DERIVATION: j = h_read_env(i,chase_uid(c))

EXCEPTION: INVALID_i(i,c)

HIDDEN V-FUNCTION: ce = h_current_env_cap(up)

PURPOSE: The value, for the process up of the capability for the current environment.

INITIALLY: ∀ u: UNDEFINED

HIDDEN V-FUNCTION: b = h_env_cap(u,up)

PURPOSE: Is u a valid environment uid of the process up?

INITIALLY: ∀(u,up): FALSE

HIDDEN V-FUNCTION: f = h_entry_point_env(u)

PURPOSE: The entry point in the "call" when the environment with uid u was created

INITIALLY: ∀(u,up): UNDEFINED

HIDDEN V-FUNCTION: f = h_return_address_env(u)

PURPOSE: The return address specified when the environment with uid u was created

INITIALLY: ∀(u,up): UNDEFINED

HIDDEN V-FUNCTION: c = h_previous_env_cap(u)

PURPOSE: The capability for the environment from within which the environment with uid u was created

INITIALLY: ∀(u,up): UNDEFINED

HIDDEN V-FUNCTION: {u} = monitor_set

PURPOSE: Set of uids for valid monitors

INITIALLY: EMPTY

HIDDEN V-FUNCTION: n = conditions(m)

PURPOSE: Number of condition variables for monitor with uid m

INITIALLY: ∀m: UNDEFINED

HIDDEN V-FUNCTION: {u} = waiting_on_monitor(m)

PURPOSE: Set of process identifiers of processes waiting on the monitor with uid m.

INITIALLY: ∀m: UNDEFINED

HIDDEN V-FUNCTION: {u} = waiting_on_condition(m,cv)

PURPOSE: Set of process ids of processes waiting on the condition cv of the monitor m.

INITIALLY: ∀(m,cv): UNDEFINED

DERIVED V-FUNCTION: {u} = blocked

PURPOSE: Set of process ids of process waiting on **any** monitor or condition variable

```
DERIVATION:  {u: ∃m(u ε waiting_on_monitor(m)
                      ∨ ∃cv(u ε waiting_on_condition(m,cv)))}
```

HIDDEN V-FUNCTION: u = in_monitor(m)

PURPOSE: The process id of the process currently in monitor m

INITIALLY: ∀m: UNDEFINED

DERIVED V-FUNCTION: b = monitor_busy(m)

PURPOSE: Is there a process in monitor m

DERIVATION: IF (in_monitor(m) = UNDEFINED) THEN TRUE ELSE FALSE

EFFECT MACRO UNBLOCK(u) /in "stop" and "delete"/

```
IF ∃m(mε'monitor_set' ∧ u ε'waiting_on_M'(m))THEN
   waiting_on_M(m) = * - u;
IF ∃m,cv(m ε 'monitor_set' ∧ cv ε N_CV(m) ∧
         u ε 'waiting_on_CV'(m,cv)) THEN
   waiting_on_CV(m,cv) = * - u;
```

OV-FUNCTION: c = create_process(st)

PURPOSE: To create a process, with initial state st, and to return its capability

EXCEPTIONS: NO_SPACE

```
EFFECT:   CHOOSE c ∋ uid(c) = u
                ∧[known_process_set = * + u;
                  state(u) = st;]
          return = c
```

O-FUNCTION: start(c)

PURPOSE: Makes schedulable, the known_process identified by the capability c

```
EXCEPTIONS:  UNKNOWN(c);
             SCHEDULABLE(c);
             NO_ABILITY(c,"start");
```

```
EFFECT:   schedulable_process_set = * + chase_uid(c)
```

O-FUNCTION: stop(c)

PURPOSE: To make the schedulable process identified by the capability c, unschedulable but still known. If a process is blocked, it becomes unblocked.

```
EXCEPTIONS:  UNKNOWN(c);
             UNSCHEDULABLE(c);
             NO_ABILITY(c,"stop");
```

```
EFFECTS:   UNBLOCK(chase_uid(c));             /see macro_specification/
           schedulable_process_set = * - chase_uid(c)
```

O-FUNCTION: delete_process(c)

PURPOSE: To delete the process identified by the capability c

```
EXCEPTIONS:  UNKNOWN(c);
             NO_ABILITY(c,"delete");
```

```
EFFECT:   UNBLOCK(c);                         /see macro_specification/
          IF chase_uid(c) ∈ schedulable_process_set THEN
            schedulable_process_set = * - chase_uid(c);
          Known_process_set = * - chase_uid(c);
```

O-FUNCTION: call(f,n,⟨ce⟩,⟨up⟩)

PURPOSE: To create a new environment, with the topmost n elements of the current environment as the initial n elements of the new environment. The topmost n elements are deleted from the current environment. The capability for the entry point address is f. ⟨up⟩ is the uid of the process, and is an implicit parameter. ⟨ce⟩ is the capability for the current environment, and is also implicit.

```
EXCEPTIONS:  INVALID_i(n,ce);
             NO_ABILITY(f), "call");
             INVALID_OFFSET(f);

EFFECT:  CHOOSE c [uid(c) = u] ∧
          [h_env_cap(u,up) = true;
           h_current_env_cap(up) = c;
           h_env_length(u) = n;
           h_env_length(ue) = * - n;
           h_entry_point_env(u) = f
           h_return_address_env(u) = 'program_counter(up) + 1;
           ∀i(0 < i ≤ n)h_read_env(i,u) = h_read_env('h_env_length'(ue) - n+i,ue)
             h_previous_env_cap(u) = ce;
             program_counter(up) = f ]
```

O-FUNCTION: return(n,⟨ce⟩,⟨up⟩)

PURPOSE: To return to the previous environment, to append the initial n elements of the current environment to the previous environment. ⟨ce⟩ is the capability for the current environment and ⟨up⟩ is the uid for the process. These are implicit parameters.

```
EXCEPTION:  INVALID_i(ce,n);

EFFECT:  h_env_cap(ue,up) = FALSE;
         h_current_env_cap(up) = 'h_previous_env_cap'(ue)
         program_counter(up) = 'h_return_address_env'(ue)

          LET 'h_previous_env_cap'(ue) = c

         h_env_length(u) = * + n
         ∀i(0 < i ≤ n) h_read_env('h_env_length'(u) + i,u) =
           h_read_env(i,ue);
```

O-FUNCTION: push(f,n,⟨ce⟩)

PURPOSE: To append n elements starting at location f to the current environment,⟨ce⟩, of the process identified by the uid up.

```
EXCEPTIONS:  INVALID_n:
             NO_ABILITY(f, read)
             OVERFLOW(offset(f + n));
             OVERFLOW_env(ce,n)

EFFECT:  ∀i(0 < i ≤ n)[n_read_env(h_env_length(ue) + i,ue) = read(f + i)];
         h_env_length(ue) = * + n;
```

O-FUNCTION: pop(f,n,⟨ce⟩

PURPOSE: To transfer the contents of the top n elements of the current environment to the n locations beginning at f the length of the current environment is decreased by n.

EXCEPTIONS: INVALID_i(ce,n) ¬

NO_ABILITY(f,write)

OFFSET_ERROR(f + n);

EFFECT: env_length(ce) = 'env_length(ce)' - n;

$\forall i(0 < i \leq n)$[read(f + i) = 'h_read_env'('h_env_length(ue) - i,ue)];

O-FUNCTION: write_out_of_env(f,i,⟨ce⟩

PURPOSE: To write the i^{th} element of the current environment ce into the location f

EXCEPTIONS: INVALID_i(i,ce)

NO_ABILITY(f,"write")

OFFSET_ERROR(f)

EFFECT: h_read(f) = 'h_read_env'(i,ue)

O-FUNCTION: write_into_env(f,i,⟨ce⟩

PURPOSE: To write the contents of location f into the i^{th} element of the current environment ⟨ce⟩

EXCEPTIONS: INVALID_i(i,ce)

NO_ABILITY(f,"read")

OFFSET_ERROR(f)

EFFECT: h_read_env(i,ue) = read(f)

O-FUNCTION: set_upper_bound(n,⟨ce⟩

PURPOSE: To decrease the length of the current environment ce by n

EXCEPTION: INVALID_i(n,ce)

EFFECT: h_env_length(ue) = * - n;

O-FUNCTION: write_env(f,i,c)

PURPOSE: To write the contents of the location represented by the offset capability f, into the i^{th} element of the environment c. The process to which the environment c belongs must have been stopped.

EXCEPTIONS: INVALID_ENV_CAP(c);

INVALID_i(i,c)

NO_ABILITY(f, "read")

NO_ABILITY(c,"write")

OFFSET_ERROR(f)

EFFECT: h_read_env(i,u) = read(f)

```
O-FUNCTION:  truncate_env(c,n)

PURPOSE:  To decrease the length of the environment c, by n elements.
          The topmost n elements are made undefined.  The process to
          which the environment c belongs must have been stopped.

EXCEPTIONS:  INVALID_ENV_CAP(c)
             INVALID_i(n,c)
             NO_ABILITY(c,"delete")

EFFECT:  h_env_length(u) = * - n;
```

```
O-FUNCTION:  append_env(f,c)

PURPOSE:  To append the contents of the location represented f to the
          environment with capability c.  The process to which the
          environment belongs must be unschedulable but known, i.e. stopped.

EXCEPTIONS:  INVALID_ENV_CAP(c)
             OVERFLOW_ENV(c,1)
             NO_ABILITY(e(f),"read")
             OFFSET_ERROR(e(f))
             NO_ABILITY(c,"append")

EFFECT:  h_env_length(u) = * + 1;
         h_read_env('h_env_length'(u) + 1,u) = read(e(f));
```

```
OV-FUNCTION:  c =  create_monitor(cm,n)

PURPOSE:  To create a monitor with n condition variables and return
          the capability for that monitor

EXCEPTIONS:  TOO_MANY_MONITORS
             INVALID_CV(n)

EFFECT:  CHOOSE c ∋ uid(c) = u ∧ abilities(c) = ALL
              ∧

                    monitor_set = * + u;
                    conditions(u) = n;
                    waiting_on_monitor(u) = EMPTY;
                    ∀i(1 < i < n) (waiting_on_condition(u,i) = EMPTY);
value: c            in_monitor(u) = UNDEFINED]
```

O-FUNCTION: delete_monitor(c)

PURPOSE: Deletes monitor with capability c, only if there is no process in it and there are no processes waiting on any condition inside the monitor.

```
EXCEPTIONS:  INVALID_MONITOR(c)
             MONITOR_BUSY(c)
             BLOCKED_PROCESSES(c)
             NO_ABILITY(c,"delete")

EFFECT:  monitor_set = * - u;
         conditions(u) = UNDEFINED;
         waiting_on_monitor(u) = UNDEFINED;
         ∀i(1 ≤ i ≤ 'conditions'(u)): [waiting_on_condition(u,i) = UNDEFINED]
```

O-FUNCTION: enter_monitor(c,⟨cp⟩)

PURPOSE: Process up (cp is an implicit parameter) desires access to the monitor with capability c

```
EXCEPTIONS:  INVALID_MONITOR(c)

EFFECT:  IF ¬ 'monitor_busy'(u) THEN in_monitor(u) = up
           ELSE waiting_on_monitor(u) = * + up
```

O-FUNCTION: exit_monitor(c,⟨cp⟩)

PURPOSE: Called by process up,exiting from the monitor with capability c

```
EXCEPTIONS:  INVALID_MONITOR(c)
             INVALID_MONITOR_PROCESS(c,cp)
                              monitor
EFFECT:  IF waiting_on_ (u) = EMPTY
           THEN in_monitor(u) = UNDEFINED
           ELSE [CHOOSE k∋ k ∈ 'waiting_on_monitor'(u)
                             ∧ [waiting_on_monitor(u) = * - k;
                                in_monitor(u) = k ]]
```

O-FUNCTION: wait(c,cv,⟨cp⟩)

PURPOSE: Process up waits on condition cv of the monitor with capability c. Process up must have been inside the monitor. Access to the monitor is released

```
EXCEPTIONS:  INVALID_MONITOR(c)
             INVALID_MONITOR_PROCESS(c,cp)
             INVALID_CONDITION(c,cv)

EFFECT:  waiting_on_condition(u,cv) = * + up;
         IF waiting_on_monitor(u) = EMPTY
           THEN in_monitor(u) = UNDEFINED
           ELSE [CHOOSE k∋ k∈ 'waiting_on_monitor'(u)
                             ∧ [waiting_on_monitor(u) = * - k
                                in_monitor(u) = k]]
```

O-FUNCTION: signal(c,cv,(cp))

PURPOSE: Process up, signals condition cv inside monitor u. One of the processes, if any, waiting on that condition is unblocked and given access to the monitor u. Process up is added to waiting_on_monitor(u).

EXCEPTIONS:

```
INVALID_MONITOR(c)
INVALID_MONITOR_PROCESS(c, cp)
INVALID_CONDITION(c,cv)
```

EFFECT:

```
IF ¬ (waiting_on_condition(u,cv) = EMPTY)
  THEN [CHOOSE k ∋ k ∈ 'waiting_on_condition(u,cv)
                   ¬ [waiting_on_condition(u,cv) = * - k
                      in_monitor(u) = k
                      waiting_on_monitor(u) = * + up]]
```

Appendix B

DATA REPRESENTATIONS

This appendix discusses the implementation of the system, illustrating the use of the methodology. In particular, Section B.1 gives a representation of a directory object as an extended-type object (implemented out of a list-linked segment). Section B.2 gives representations for level 5 objects and segments.

B.1 Representation of Directories

In this section we illustrate Stage 3 of the design, implementation, and proof methodology for our system. In particular, we give a representation of each directory in terms of a segment. This representation is formally described by mapping function expressions that associate V-functions of the directory module with expressions containing V-functions of the extended-type module and of the segment module. We give an informal proof of the consistency of these mapping function expressions.

Figure B.1 displays the representation for a particular directory in terms of a segment. For ease of description here we have eliminated from the directory module the lock list and distinguished entry functions. The following decisions are embodied in this representation:

- The symbolic name and the capability associated with each entry occupy adjacent positions. For simplicity we assume here that a name can be no longer than one machine word.
- The segment positions storing entry information are linked together. The position following that for the capability

holds the starting address of the next entry. For the last entry, the value in this link position is 0.

- The link to the first entry is in position 0 of the segment.

There are several decisions regarding the representation that need not be made at this stage, and that can be postponed to Stage 4 (the abstract implementation stage). Among such decisions are the following:

- Ordering of the entries within a segment--the programs that form the directory manager can order the entries in a particular way (e.g., sorted according to symbolic name).

- Free space management--within any given segment there might be unused positions that correspond to deleted entries. It is possible to link these positions so that they are available for newly created entries. Another (not very efficient) possibility would be to have the directory manager carry out a garbage-collection after each entry deletion.

As indicated in Chapter 3, we are using mapping function expressions to characterize the decisions concerning directory representation that are to be made at Stage 3. Since the state of the directory module is completely defined by the values of the two HIDDEN V-functions "h_get_cap" and "h_valid_dir", we need only write mapping function expressions for these functions. Table B.1 gives the mapping function expressions that characterize the representation of Figure B.1.

We wish to write mapping function expressions to yield a defined value for a V-function over the domain for which the V-function is defined, and to yield UNDEFINED otherwise. Consider the representation of "h_get_cap(u,n)", whose purpose is to return the capability associated with a symbolic name n in directory with uid u. The function "h_get_cap(u,n)" is defined under the following conditions: (1) an object with uid u is known to exist, (2) the particular object with uid u is of type "directory," and (3) there exists an entry in the directory with symbolic name n.

In the mapping function expression as written in Table B.1, the three conditions mentioned above have been transformed to V-functions of the extended-type module and the segment module. The conditions (1) and (2) are written directly in terms of the appropriate extended-type HIDDEN V-functions. Condition (3) requires that the extended-type manager associates a segment capability s (having uid u1) with the directory (with uid u). Within this segment there must be a position x, that is linked to position 0, and that contains n. The auxiliary function "address (n2,u2)" returns the displacement of n2 in the segment, if n2 is within the linked portion of the segment, and 0 otherwise. If each condition for this mapping function expression evaluates to TRUE, then h_get_cap(u,n) is defined, and the associated capability is found in position x + 1 of the segment with uid u1.

We now present informal arguments that the two mapping function expressions are consistent with respect to the specifications of the three modules. That is, it must be shown that

(1) All defined states of the directory module map down to states of the segment and extended-type modules;

(2) Let S1 and S2 be states of the directory module, T1 and T2 be states of the extended-type module, and W1 and W2 be states of the segment module. If (T1,W1) is an image of S1, and if (T2,W2) is an image of S2, and if S1 ≠ S2, then (T1,W1) ≠ (T1,W2).

Property 1 is satisfied easily since the mapping function expressions yield an UNDEFINED value only for directory V-function values that correspond to exception conditions for the V-functions. The proof for property 2 requires two steps. First two distinct directories, with distinct uid's u and u', will have different segment capabilities, since the two segment capabilities are represented in the extended-type module as s = h_impl_cap(u) and s' = h_impl_cap(u'). From the specifications of the extended-type module, it is impossible for two directories with different uids to have the same segment capability, since such capabilities are created by calling the function "initialize (d,cdt,⟨st⟩)" separately for each of the directories. Here d is a directory capability, cdt is the type manager's capability for directories, and st is the capability for creating segments. Second, it remains to show that for a given directory uid u, distinct sets of entries are transformed to distinct values in the linked segment positions. However, this is obvious from the mapping function expression for "h_get_cap(u,n)", since the value of each symbolic name and its associated capability appear respectively as the contents of the first and second segment positions pointed to by some link.

The abstract implementations of the directory functions and their proofs of correctness with respect to mapped specifications are not included here. These programs are not conceptually complex, but they require the binding of additional design decisions. For example, the implementation of "create_dir" will involve successive calls by the directory manager on the extended-type module functions: (1) "create_object," to create a capability for a new object of type "directory"; (2) "initialize", to give the directory a segment implementation; (3) "change_seg_size", to make the segment length at least 1 (this initializes position 0 of the segment to 0, indicating a null list). The implementation of the directory O-function "remove_entry(d,n)" would obviously search for the symbolic name n in the segment corresponding to d. The program would then modify the link pointing to n to point to the successor (if one exists) of this entry. If we decided to have a "free-list" within each segment, then the three segment positions associated with n would become zeroed out and attached to the free-list. Note that the assertions for this program are not explicitly concerned with the free-list--just the values of segment positions linked in the list of entries--unless we extend them to refer to segment utilization.

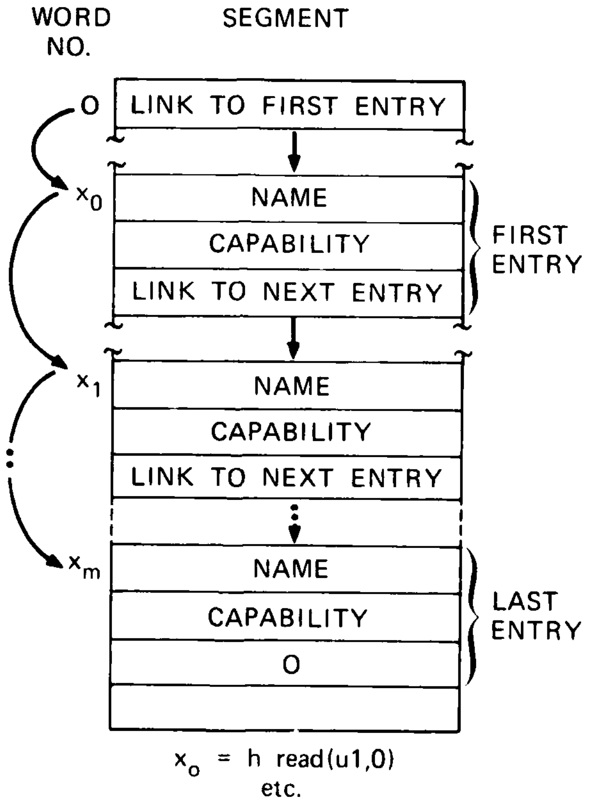
Initialization is treated briefly in Chapter 10. At level 6, the directory module can initialize itself by calling the OV-function "cd = create_type(ct)" of the extended-type manager. The parameter ct indicates that the caller wishes to establish a new type, in which case the returned capability cd is the type manager's capability for directories. By calling this function, the directory manager makes itself known to the extended-type manager.

This discussion describes the formal basis on which many properties of the implementation of the system can be stated and proved. The mapping functions correspond to a "visualized implementation" as shown in Figure B.1. The remaining details need be bound only at the final coding stage.

B.2 Representation of Segments and Extended Type Objects

The level 4 mapping functions are given in terms of level 3 h_read and the functions at levels 1 and 0. The data local to level 4 is assumed to be organized in four segments. The segment with uid s1 contains the list of all virgin segment uids, and their size. The segment with uid s2 contains the location of all pages of all segments. The segment with uid s3 contains the list of all revocable uids and the segment with uid s4 their linktuples. The segments s1, s2, s3 and s4 are maintained by level 3. That is, the segment operations applicable to segments s1, s2, s3 and s4 are level 3 operations. The data structures for these segments are shown in Figures B2 and B3.

We will define four auxiliary functions, list_1, list_2, list_3, and list_4, list_i being defined on segment si. Given the offset for an element of a list, these functions return the offsets for the remaining elements of the list. The functions are defined below. Note that the function "h_read" used is the level 3 function "h_read(u,j)".



SA-2581-13

FIGURE B.1 REPRESENTATION OF A DIRECTORY AS A SEGMENT WITH UID u

Table B.1

MAPPING FUNCTIONS FOR DIRECTORIES

Auxiliary function definitions:

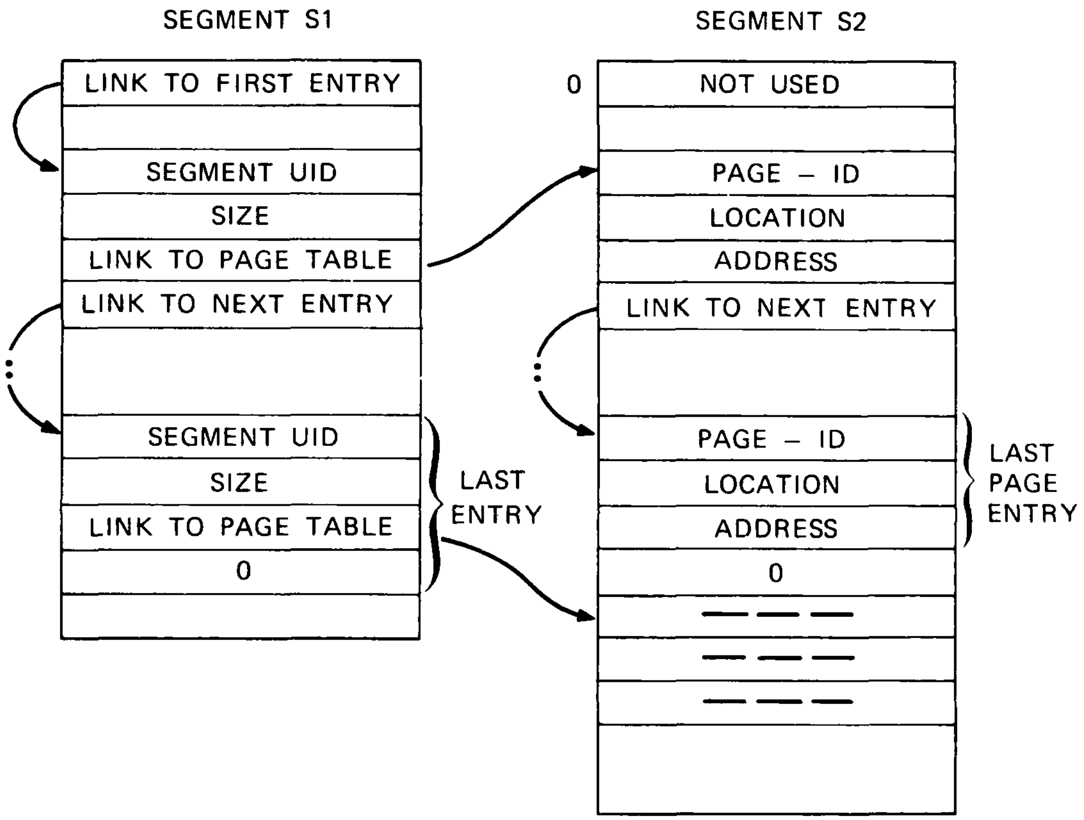
```
address(n2,u2) = aux_address(n2,u2,0)
aux_address(n3,u3,x3) =
  IF h_read(u3,x3) = 0 THEN 0
  ELSE IF h_seg_size(u3) <
          h_read(u3,x3) + 2
       THEN ERROR
  ELSE IF h_read(u3,h_read(u3,x3)) = n3
       THEN h_read(u3,x3)
  ELSE aux_address(n3,u3,h_read(u3,x3) + 2)
```

/Note: address(n2,u2) returns the displacement of entry n2 in directory u2 if it exists, otherwise 0; aux_address is a recursive function that does the same thing, starting the search at displacement 0 in segment u3./

Mapping function expressions:

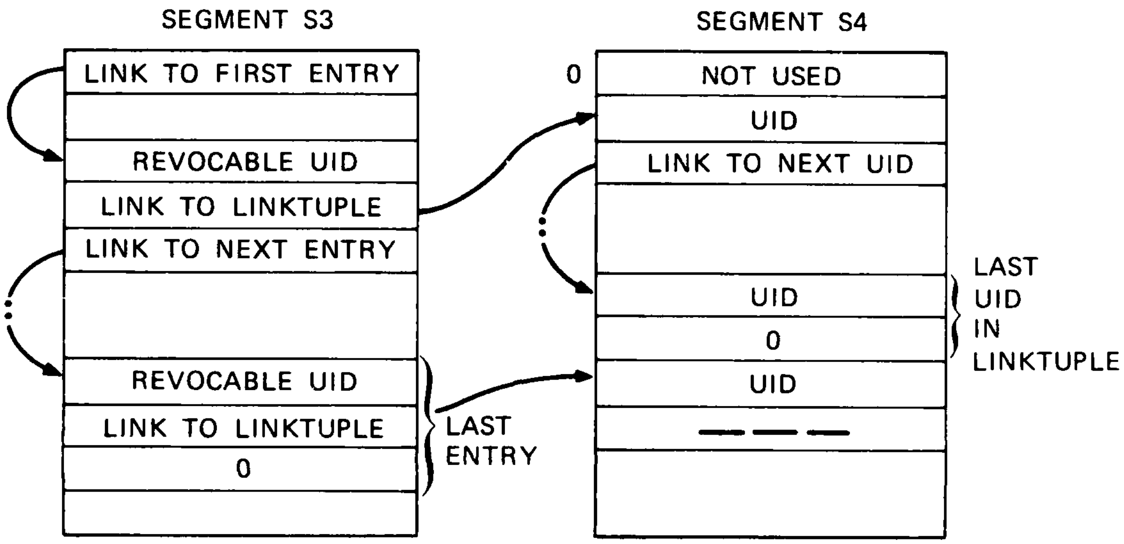
```
h_get_cap(u,n):                                   /level 6/
  IF h_object_exists(u) ∧ h_get_type(u) = "dir"   /level 5/
      ∧ ∃x,ul,s
        ∧ (s = h_impl_cap(u)
          ∧ ul = get_uid(s)
          ∧ x > 0
          ∧ h_seg_size(ul) ≥ x+2                  /level 4/
          ∧ address(n,ul) = x)
  THEN h_read(ul,x+1)
  ELSE UNDEFINED

h_valid_dir(u): h_object_exists(u)
        ∧ h_get_type(u) = "dir"
        ∧ ∃s(s = h_impl_cap(u))
```



SA-2581-14

FIGURE B.2 DATA STRUCTURES FOR S1 AND S2



SA-2581-15

FIGURE B.3 DATA STRUCTURES FOR S3 AND S4

Table B.2

MAPPING FUNCTIONS FOR SEGMENTS

Auxiliary Function Definitions

```
list_1(s1,x):  IF x = 0 THEN[IF h_read(s1,x) = 0 THEN EMPTY
                           ELSE h_read(s1,x) ∪ list_1(s1,h_read(s1,x))]
               ELSE[IF h_read(s1,x+3) = 0 THEN EMPTY
                    ELSE h_read(s1,x+3) ∪ list-1(s1,h_read(s1,x+3)]

list_2(:2,x):           IF x = 0 THEN EMPTY ELSE x ∪ list_2(h_read(s2,x+3))
list_3(s3,x):           IF x = 0 THEN [IF h_read(s3,x) = 0 THEN EMPTY
                                       ELSE h_read(s3,x) ∪ list_3 (h_read(s3,x))
                        ELSE [IF h_read(s3,x+2) = 0 THEN EMPTY
                              ELSE h_read(s3,x+2) ∪ list_3(s3,h_read(s3,x+2))
list_4(s4,x):           IF x = 0 THEN EMPTY ELSE x ∪ list_4(s4,h_read(s4,x+1))
/Initially h_read(s1,0) = 0 and h_read(s3,0) = 0./
```

Mapping Function Expressions

```
{u} = virgin_set:       {u | ∃x(x ε list_1(s1,0) ∧ h_read(s1,x) = u)}
{u} = revocable_set:    {u | ∃x(x ε list_3(s3,0) ∧ h_read(s3,x) = u)}

[u] = linktuple(u1):    IF u1 ε virgin_set THEN {u1}
                        ELSE (IF u1 ε revocable_set THEN
                              (IF x ε list_3(s3,0) ∧ h_read(s3,x) = u1
                               THEN[u1 , list_4(s4,h_read(s3,x+1))])
                               ELSE UNDEFINED)
                             ELSE UNDEFINED

i = h_size(u):          IF u ε virgin_set THEN
                           [IF x ε list_1(s1,0) ∧ h_read(s1,x) = u
                            THEN h_read(s1,x+1) ELSE UNDEFINED]
                        ELSE UNDEFINED

/page_no(j):            is a function which returns the page_uid corresponding
                        to the segment offset j ./
/displacement(j):       is a function which returns the displacement within
                        the page for the segment offset j./
```

Table B.2 (Concluded)

```
i = h_read(u,j):           /Note that this is the level_4 h_read being defined,
                           using the levels 3, 1, and 0 h_read functions /

                           IF (u||page_no(j) ε adr_map ∧ displacement(j)
                               ≤ bounds (u||page_no(j))
                           THEN h_read(entry_adr_map(u||page_no(j))||displacement(j))
                           ELSE

                           IF u ε virgin_set THEN
                              [IF j ≤ h_size(u) THEN disk_read(secondary_address(u,j))]
                                                 ELSE UNDEFINED


secondary_address(u,j):     (h_read(s2,y+2) + displacement(j))|
                           [(h_read(s2,y) = page_no(j))
                           ∧(y ε list_2(s2,x)]|
                               [(h_read(s1,z+2) = x)
                               ∧(z ε list_1(s1,0)) ∧ (h_read(s1,z) = u)])]

disk_read(secondary_address:   the contents of the disk location corresponding
                               to the secondary address.
```

Level 5: Mapping Function Expressions

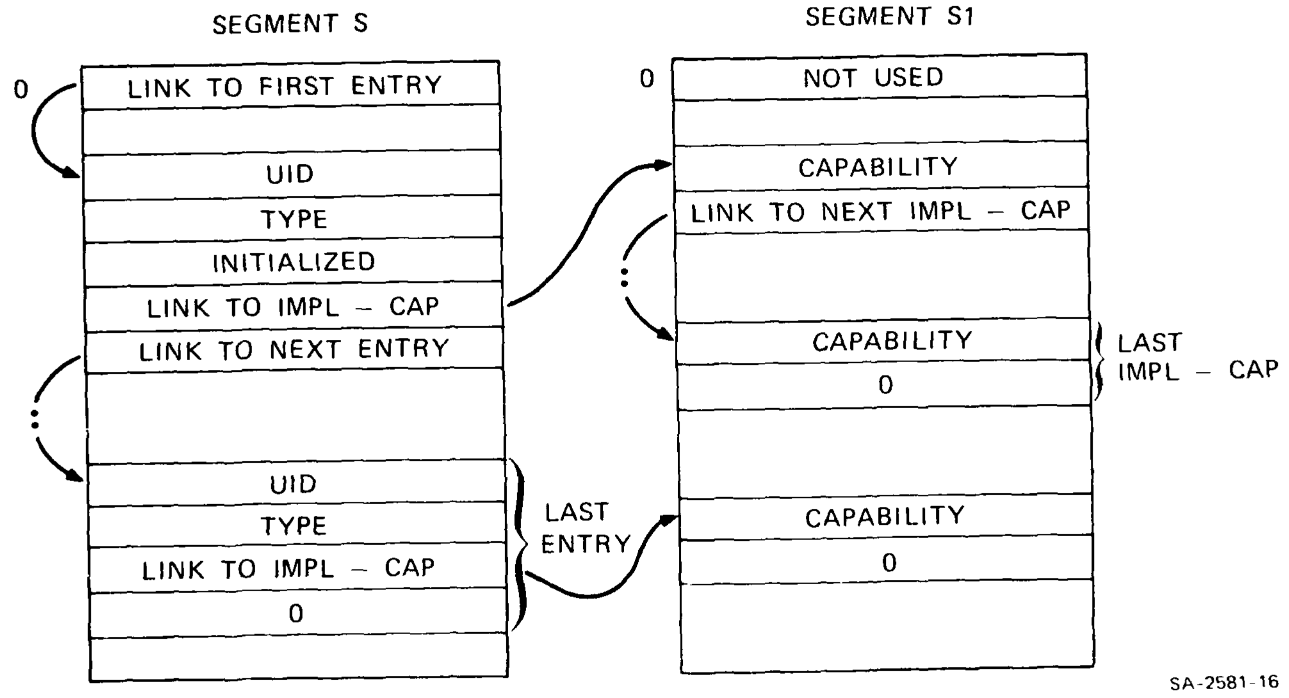
The data structures for the segments used by level 5 are shown in Figure B.4. The mapping function expressions for level 5 are given in Table B.3 in terms of level 4 functions and the auxiliary functions list(s,x), list_l(sl,x) and tuple(x). The segment with uid s is used for maintaining information about all objects and the segment with uid sl is used for maintaining their implementation capabilities.

The function list(s,x) returns the offsets of entries for all objects in the list beginning from offset x. The value 0 is used for indicating the end of a list. The function list_l(s,x) is similar to list(s,x). It returns the list of implementation capabilities.

Table B.3

LEVEL 5: MAPPING FUNCTION EXPRESSIONS

```
b   = h_exists(u):  IF ∃x(x ∈ list(s,u) ∧ h_read(s,x) = u)
                     THEN TRUE ELSE FALSE
i   = h_object_type(u): IF ∃x(x ∈ list(s,0) ∧ h_read(smx) = u
                            THEN h_read(s,x+1) ELSE UNDEFINED
b   = h_initialized(u): IF ∃x(x ∈ list(s,0) ∧ h_read(s,x) = u) THEN h_read(s,x+2)
                            ELSE FALSE
{c} = h_impl_cap(u):    IF ∃x(x ∈ list(s,0) ∧ h_read(s,x) = u) THEN
                           tuple(h_read(s,x+3)
list(s,x):      IF x = 0 THEN[IF h_read(s,x) = 0 THEN EMPTY
                              ELSE x ∪ list(s,h_read(s,x))]
                ELSE IF h_read(s,x+4) = 0 THEN EMPTY
                ELSE h_read(s,x+4) ∪ list(s,h_read(s,x+4)
tuple(x) =      IF x = 0 THEN EMPTY ELSE
                {h_read(sl,y) | y ∈ x ∪ list_l(sl,x)}
list_l(sl,x):   IF h_read(sl,x+1) = 0 THEN EMPTY ELSE
                   h_read(sl,x+1) ∪ list_l(sl,h_read(sl,x+1))
```



SA-2581-16

FIGURE B.4 DATA STRUCTURFS FOR LEVFL 5 SEGMENTS

Appendix C

THE SECURE DOCUMENT MANAGER

This appendix discusses briefly two models for computer representation of military security. It also presents specifications for a particular approach to realizing such a model for military security, illustrating the utility of the methodology. Two potential implementations are sketched, illustrating the utility of the operating system. The contents of this appendix are considered to be preliminary.

Existing Security Models

The two models discussed here are the Weissman model, used in the ADEPT-50 time-sharing system (Weissman [69]) and the Bell and LaPadula model (Bell and LaPadula [74]) being used at MITRE for implementation on a PDP-11 (and for retrofitting into Multics). In the long run, neither model is really adequate; however, they are representative of what exists today.

ADEPT-50

The security controls in the ADEPT-50 time sharing system (C. Weissman [69]) are authority based, as opposed to capability based. ADEPT-50 supports users of different clearance levels (top-secret, secret, confidential, and unclassified) and each is able to operate within a set of categories. "Need-to-know" is implemented by the granting of access rights. An access list is maintained with each object, which contains the list of users with need-to-know.

Four types of access are implemented: "read," "write(append)," "read and write," and "read and write with lockout override." The delegation and revocation of these access rights is restricted to the owner. A catalogue of objects is maintained but is not accessible to the users. The access lists are maintained with the objects.

There is no concept of confinement, and it is possible for a user of higher clearance to transmit information intentionally or unintentionally to users of lower classification. Also, programs of different classification operate at the clearance level of the user (job), and this can compromise security (e.g., presenting the opportunity for a Trojan horse).

New objects created are classified at the same level as the current clearance level of the user (job).

The Bell and LaPadula Model (MITRE)

The MITRE model attempts to remove some of the disadvantages of the Weissman Model, such as the confinement problem and delegation of access rights.

The confinement problem is attacked by not permitting any information to flow to a user with lower clearance from a user who currently has access to higher-level classified information. This includes writing into files that may be read by users of lower classification, granting and denying access rights to objects of lower classification, as well as creating and deleting such objects.

The right to delegate access rights is implicit. All documents are catalogued in directories (which form a hierarchy). The possession of a "write" access to a directory also gives the right to delegate and revoke access to objects in the directory.

Delegation of access rights to more than one level is very difficult (especiallv with revocation), since it is not easy to determine the chain of delegated rights. Also, delegation is on a per-directory basis for multi-level delegation.

The MITRE Model supports the following access rights: "execute," "read," "append," and "write." Their "append" is different from ADEPT's "append," since it permits overwriting. Also, need-to-know-right to append is sufficient for append access to objects of higher classification.

All new objects created have the same classification as the current clearance of the user. A user cannot delete, or rescind rights to objects for which it does not currently possess potential write access (i.e., the objects having the same clearance level as the current clearance of the user). (It is assumed that the system will prevent a user from exercising an access right for an object o, unless the object is in the access set of the process for that access right.) New objects created have a clearance level of the creator. This is controlled implicitly by requiring the object to be catalogued in a directory, with the restriction that all objects in a directory have clearance levels greater than or equal to the clearance level of the directory.

The SRI Model

The model for a classified document manager discussed next is essentially the same as the MITRE model. The only difference is that it does not explicitly maintain the tree structure for directories, and "append" access rights are not needed. At this point we restrict the classified documents to be segments, just as in the MITRE model. It seems straightforward, however, to extend the model to include sharing of classified objects other than segments. We feel that this will be important in achieving general applicability.

Thus in this model each subject or object has a classification (confidential, secret, etc.) and a set of mutually exclusive categories.

Subjects also have a need-to-know. A necessary condition (i.e., a clearance condition) for a subject to gain access to an object is that the subject's clearance level be higher than or equal to the object's clearance level. Here the clearance level is defined as the two-tuple [classification, category-set]. The clearance level of a subject is at least as great as the clearance level of the object if and only if

subject's classification $\geq$ object's classification AND
subject's category-set $\supseteq$ object's category-set.

A subject's clearance level is equal to an object's clearance level iff

subject's classification = object's classification AND
subject's category-set $\equiv$ object's category-set.

At this point we are considering only segments as classified documents. We define six access rights for segments, the right to "write" which allows altering the contents of the segment, the right to "read" which allows reading the contents of the segment, the right to "execute" which allows the contents of the segment to be interpreted as instructions to a processor, the rights to "grant" and "rescind" access rights to other subjects, and the right to "delete."

Another necessary condition (right's condition) for a subject to gain a "write," "read" and/or "execute" access to an object is that the subject should possess the necessary right for that object, representing the appropriate need-to-know.

A third necessary condition (non-compromising condition) is that the subject is granted "write" access for an object iff

the clearance level of the subject = clearance level of the object.

With respect to "write" access, this condition by itself is equivalent to the union of the "*-property" of Bell and LaPadula and the clearance condition above. The satisfaction of these three necessary conditions is sufficient for a subject to gain access to an object.

Each subject is given a limiting clearance level, up to which he may operate. The clearance level at which the subject is operating is known as the subject's current clearance level. The necessary conditions to be satisfied in granting the requests are evaluated in terms of the subject's current clearance level.

Specifications for this secure document manager (SDM) are given in Table C.1. Two implementations of this SDM are outlined below. The first uses linkage sections to prevent any direct use of capabilities. The second forces copying of any object to be shared for reading only, after first removing from it all capabilities.

Implementation of a Secure Document Manager Based on Linkage Sections--Outline

1. The SDM is a subsystem available only via a special login.

2. All objects that are to be accessed or created while operating under the SDM are done through the SDM. The accessing (creating) process gets an extended object's capability which can be used only via the SDM. The SDM will map the user-provided capabilities onto object capabilities.

3. A user may create objects with the same clearance level as its current clearance level.

4. A user may not have any access to a document or object of a higher classification (the clearance condition).

5. A user may have only "read" or "execute" access to objects of lower classification (the complement of the non-compromising condition).

6. All access to secure objects is via the linkage sections. The assumption that capabilities may not be read out of linkage sections is critical here. (This is consistent with Section A.8.)

7. All segment creation is done through the SDM.

8. A user does not have a capability for his linkage section.

Implementation of a Secure Document Manager Based on Restricted Capabilities--Outline

1. The SDM is a subsystem available only via a special login.

2. Unrestricted (direct) communication is allowed only between users with exactly the same current classification and current categories.

3. The SDM keeps copies only of read-only, execute-only, and read or execute-only objects; i.e., all capabilities have only these access bits set. Further, no user has any other capability for this object.

4. Users may ask for access to these objects by symbolic name. The SDM returns the appropriate capability.

5. A user may create objects of exactly its own classification and category.

6. A user may delete objects, but an object is deleted only when no user has access to it. A user is not given access to an object declared to be deleted.

7. The access rights to an object may be granted or rescinded at any time, but have affect only when a process requests access. (Rescinding is not possible for objects for which a user has current access.)

Efficiency of Implementations

It appears that the operating system provides suitable support to permit efficient implementation of the specifications of Table C.1. More work is needed to refine the design and to explore alternative implementations.

The Login Process

When a user wishes to work under a security environment, the "login" command (Chapter 6) checks for appropriate identification and authorization. A user may operate at any clearance level up to his limiting clearance level. The current clearance level has to be declared at log-in, which establishes a process for the user with the current clearance level, and gives it access to the user's directory with the current clearance level, as well as access to the SDM and other necessary system functions.

Communication Set-Up

Two users with the same current clearance level may set up a communication link by sharing a common entry in their directories. The function "send_link (directory name, entry name, user_id_send)" enables a user with id user_id_send to send the capability corresponding to the entry to a user with user identification "user_id_rec." The user with id "user_id_rec" may obtain the link by "receive_link (directory name, entry name, user_id_send)," whereupon the SDM puts the capability in the specified entry.

Table C.1

FUNCTIONS OF THE SDM

V-Functions			O-Functions
i	=	classification(o)	get_access(o,p,"ar")
{ct}	=	category_set(o)	release_access(o,p,"ar")
{ar}	=	access_rights(p,o)	grant_access(o,pd,pa,"ar")
i	=	current_classification(p)*	rescind_access(o,pd,pa,"ar")
{ct}	=	current_category_set(p)*	create_object(p,o)
{o}	=	access_set("ar",p)	delete_object(p,o)
{o}	=	object_set	make_process_known(p)
{r}	=	process_set	make_process_unknown(p)
i	=	object_count(p)	
i	=	ref_count(p)	
p	=	creator(o)	

* set by login.

SDM PARAMETERS

o:	symbolic name	/object/
p:	symbolic name	/process/
"ar":	access right	
pd:	symbolic name	/donor process/
pa:	symbolic name	/acceptor process/
i:	integer	
ct:	category	

access_rights: {execute,read,write,grant,rescind,delete}
max_cat_set: the maximum category set
max_cl: maximum classification
init_o: symbolic names for the initial_set of objects
init_p: symbolic names for the initial_set of processes
max_count: maximum number of existing objects created by a process

EXCEPTIONS FOR SDM

INVALID_OBJECT(o): o $\notin$ object_set

INVALID_PROCESS(p): p $\notin$ process_set

VALID_PROCESS(p): p $\in$ process_set

INVALID_ACCESS_RIGHT("ar",p,o): "ar" $\notin$ access_rights(p,o)

INVALID_CLEARANCE(p,o): [classification(o) $>$ classification(p)
$\vee$ category_set(o) $\not\subseteq$ category_set(p)]

INVALID_*_PROPERTY("ar",p,o): "ar" ="write"$\wedge$ [current_classification(p) $\neq$
classification(o)
$\vee$ current_category_set(p) $\not\equiv$
category(o)]

INVALID_ACCESS_SET(o,"ar",p): o $\notin$ access_set("ar",p)

INVALID_GRANT(p,o): grant $\notin$ access_rights(p,o)
$\wedge$ INVALID_*_PROPERTY("write",p,o)

INVALID_RESCIND(p,o): rescind $\notin$ access_rights(p,o)
$\wedge$ INVALID_*_PROPERTY("write",p,o)

OVERFLOW_OBJECT_LIMIT(p): object_count(p) $\geq$ max_count

NOT_FREE_PROCESS(p): $\exists$[(o $\in$ object_set) $\wedge$ ("ar" $\in$ access_rights)]
$\ni$ o $\in$ access_set("ar",p)

SPECIFICATIONS FOR THE SDM

HIDDEN V-FUNCTION: i = classification(o)

PURPOSE: The classification of the object o

INITIALLY: for o = init_o: max_cl
$\forall$o $\neq$ init_o: UNDEFINED

EXCEPTIONS: NONE

HIDDEN V-FUNCTION: {ct} = category_set(o)

PURPOSE: The category set of the object o

INITIAL VALUE: for o = unit_o: max_cat
$\forall$o $\neq$ init_o: UNDEFINED

EXCEPTIONS: NONE

HIDDEN V-FUNCTION: {ar} = access_rights(p,o)

PURPOSE: Defines the access rights of process p for the object o

INITIALLY: ∀p,o: UNDEFINED

EXCEPTIONS: NONE

V-FUNCTION: i = current_classification(p)

PURPOSE: Current_classification of process p

INITIALLY: ∀p: UNDEFINED

EXCEPTIONS: NONE

V-FUNCTION: {ct} = current_category_set(p)

PURPOSE: Current category set of the process p

INITIALLY: ∀p: UNDEFINED

EXCEPTIONS: NONE

HIDDEN V-FUNCTION: {o} = access_set("ar",p)

PURPOSE: The set of objects for which process p currently has the access right ar

INITIALLY: ∀("ar",p): UNDEFINED

HIDDEN V-FUNCTION: {o} = object_set

PURPOSE: Set of objects known to the secure document manager

INITIALLY: {init_o}

EXCEPTIONS: NONE

HIDDEN V-FUNCTION: p = creator(o)

PURPOSE: The identity of the process which created the object o

INITIALLY: ∀o: UNDEFINED

EXCEPTIONS: NONE

HIDDEN V-FUNCTION: {p} = process_set

PURPOSE: The set of objects known as processes to the secure document manager

INITIALLY: {init_p}

EXCEPTIONS: NONE

V-FUNCTION: i = object_count(p)

PURPOSE: The number of objects created by process p that still exist

INITIALLY: ∀p: o

EXCEPTIONS: NONE

DERIVED V-FUNCTION: i = ref_count(o)

PURPOSE: Number of processes currently having some type of access
to the object o

DERIVATION: cardinality{p | o ∈ access_set("ar",p)
∧ "ar" ∈ access rights}

O-FUNCTION: get_access(o,p,"ar")

PURPOSE: To get access right ar for the object o

EXCEPTIONS: INVALID_OBJECT(o)
INVALID_PROCESS(p)
INVALID_ACCESS_RIGHT("ar",p,o)
INVALID_CLEARANCE(p,o)
INVALID_*_PROPERTY("ar",p,o)

EFFECTS: access_set("ar",p) = 'access_set'("ar",p) + o

O-FUNCTION: release_access(o,p,"ar")

PURPOSE: To release access right ar for the object o

EXCEPTIONS: INVALID_OBJECT(o)
INVALID_PROCESS(p)
INVALID_ACCESS_SET(o,"ar",p)

EFFECTS: access_set("ar",p) = * - o

O-FUNCTION: grant_access(o,pd,pa,"ar")

PURPOSE: The donor process pd, wants to give the acceptor process pa,
the right "ar" for the object o

EXCEPTIONS: INVALID_OBJECT(o)
INVALID_PROCESS(pd)
INVALID_PROCESS(pa)
INVALID_ACCESS_RIGHT("ar",pd,o)
INVALID_GRANT(pd,o)

EFFECTS: access_rights(pa,o) = * + "ar"

```
O-FUNCTION:  rescind_access(o,pd,pa,"ar")

PURPOSE:  The donor process pd wants to rescind the acceptor
          process pa's right ar to the object o

EXCEPTIONS:  INVALID_OBJECT(o)
             INVALID_PROCESS(pd)
             INVALID_PROCESS(pa)
             INVALID_ACCESS_RIGHT("ar",pd,o)
             INVALID_ACCESS_RIGHT("ar",pa,o)
             INVALID_RESCIND(pd,o)

EFFECT:  access_rights(pa,o) = * - "ar";
         IF o ε 'access_set'("ar",pa) THEN
              [access_set("ar",pa) = * - o];
```

```
O-FUNCTION:  create_object(p,o)

PURPOSE:  To create an object with name o, and to give process p
          (creating process) all the rights to the object.  The
          created object has the same classification and category
          as the process p

EXCEPTIONS:  INVALID_PROCESS(p)
             OVERFLOW_OBJECT_LIMIT(p)

EFFECT:  object_set = 'object_set' + {o};
         classification(o) = current_classification(p);
         category(o) = current_category(o);
         access_rights(p,o) = {"ar" | "ar" ε access rights};
         object_count'(p) = * + 1;
         creator(o) = p;
```

```
O-FUNCTION:  delete_object(p,o)

PURPOSE:  Process p wants to delete object o

EXCEPTIONS:  INVALID_OBJECT(o)
             INVALID_PROCESS(p)
             INVALID_ACCESS_RIGHT("delete",p,o)
             INVALID_*_PROPERTY("write",p,o)

DELAY:  UNTIL ref_count(o) = 0

EFFECTS:  object_set = * - o;
          creator(o) = UNDEFINED;
          'object_count'('creator'(o)) = * - 1;
          access_rights (p,o) = UNDEFINED;
```

```
O-FUNCTION:  make_process_known(p)

PURPOSE:  To define p as a process

EXCEPTIONS:  VALID_PROCESS(p)

EFFECTS:  process_set = * + p
```

O-FUNCTION: make_process_unknown(p)

PURPOSE: To make process p unknown

EXCEPTIONS: INVALID_PROCESS(p)
NOT_FREE_PROCESS(p)

EFFECT: process_set = * - p